Indiscrete Affairs

Volume I — Reflection and Computing

Brian Cantwell Smith

Last edited: 17 January 2010
Please do not copy or cite
Draft only (version 0.80

Faculty of Information
45 Willcocks St, Toronto
Ontario M5S 1C7 Canada
brian.cantwell.smith@utoronto.ca

— Were this page blank, that would have been unintentional —

Table of Contents

A • Distillation

1 Foundations of Computation 3

B • Reflection

2 Reflection and Semantics in a Procedural Language
a Abstracts, Preface, and Prologue 45
b Chapter One 69
3 Reflection and Semantics in Lisp 173
4 Implementation of Procedurally Reflective Languages 229
5 Varieties of Self-Reference 283

C • Computing

6 Limits of Correctness 331
7 One Hundred Billion Lines of C++ 357
8 Semantics of Clocks 373

D • Semantics

9 Linguistic and Computational Semantics 409
10 The Correspondence Continuum 433

Epilogue 499

— Were this page blank, that would have been unintentional —

Prior Publication Details (3 in books; 4 in conferences, 3 never published)

1. **Foundations of Computation:** In Matthias Scheutz (ed), *Computationalism: New Directions*, MIT Press, 2002.
2. **Reflection and Semantics in a Procedural Language:** Never published as such; doctoral dissertation, printed as Technical Report 272 of the MIT Laboratory of Computer Science, 1982.
3. **Reflection and Semantics in Lisp:** Conference: the *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages* (POPL), Salt Lake City, Utah, Jan. 1984, pp. 23–35.
4. **Implementation of Procedurally Reflective Languages:** Conference: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming.*
5. **Varieties of Self-Reference**: Conference; Joseph Y. Halpern (ed.), *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference*, Monterey, California, March 19–22. Los Altos, California: Morgan Kaufmann: 1986.
6. **Limits of Correctness:** In D. Johnson & H. Nissenbaum (eds.), *Computers, Ethics & Social Values*, Englewood Cliffs, NJ: Prentice Hall, 456–69; (iii) in Colburn, T. R., Fetzer, J. H., & Rankin T. L. (eds.), *Program Verification*, Kluwer Academic Publishers, Dordrecht/Boston/London, 1993, pp. 275–93; and (iv) in Kling, R. (ed.), *Computerization and Controversy: Value Conflicts and Social Choices* (2nd Ed.), San Diego: Academic Press, pp. 810–25
7. **One Hundred Billion Lines of C++:** Never published. An early sketch appeared in the newsletter of the Cognitive Science Program at Lehigh University in 1997.
8. **Semantics of Clocks:** In James H. Fetzer (ed.), *Aspects of Artificial Intelligence*, Kluwer 1998, pp. 3–31.
9. **Linguistic and Computational Semantics:** Conference: *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics*, Toronto, Ontario, June 1982, pp. 9–15
10. **The Correspondence Continuum:** Never published.

— Were this page blank, that would have been unintentional —

A • Introduction

— Were this page been blank, that would have been unintentional —

1 — The Foundations of Computing[†]

1 Introduction

Will computers ever be conscious? Is it appropriate—illuminating, correct, ethical—to understand people in computational terms? Will quantum, DNA, or nanocomputers require radical adjustments to our theories of computation? How will computing affect science, the arts, intellectual history?

For most of my life I have been unable to answer these questions, because I have not known what computation is. More than thirty years ago, this uncertainty led me to undertake a long-term investigation of the foundations of computer science. That study is now largely complete. My aim in this chapter is to summarise a few of its major results.

2 Project

The overall goal has been to develop a comprehensive theory of computing. Since the outset, I have assumed that such an account must meet three criteria:

1. **Empirical:** It must do justice to—by explaining, or at least supplying the wherewithal with which to explain—the full range of computational practice;

[†] Originally published in Matthias Scheutz (ed), *Computationalism: New Directions*, MIT Press, 2002. The paper is distilled from, and is intended to serve as an introduction to, a series of books that collectively report, in detail, on the investigation identified in §2. The study of computing will be published as *The Age of Significance* (Smith, forthcoming—henceforth AOS); the metaphysical territory to which that study leads is introduced in *On the Origin of Objects* (Smith 1996).

2. **Conceptual:** As far as possible, it must discharge, and at a minimum own up to, its intellectual debts (e.g., to semantics), so that we can understand what it says, where it comes from, and what it "costs"; and
3. **Cognitive:** It must provide an intelligible foundation for the computational theory of mind: the thesis, often known as *computationalism*,[1] that underlies traditional artificial intelligence and cognitive science.

The first, "empirical" requirement, of doing justice to practice, helps to keep the analysis grounded in real-world examples. By being comprehensive in scope, it stands guard against the tendency of narrowly-defined candidates to claim dominion over the whole subject matter.[2] And it is humbling, since the computer revolution so reliably adapts, expands, dodges expectations, and in general outstrips our theoretical grasp. But the criterion's primary advantage is to provide a vantage point from which to question the legitimacy of all extant theoretical perspectives. For I take it as a tenet that what Silicon Valley *treats* as computational *is* computational; to deny that would be considered sufficient grounds for rejection. But no such *a priori* commitment is given to any *story* about computation—including the widely-held recursion- or Turing-theoretic conception of computability, taught in computer science departments around the world, that currently lays claim to the title "The Theory of Computation."[3] I also reject

[1] The same thesis is sometimes referred to as *cognitivism*, though strictly speaking the term "cognitivism" denotes a more specific thesis, which takes mentation to consist in rational deliberation based on patterns of conceptualist (i.e., "cognitive") inference, reminiscent of formal logic, and usually thought to be computationally implemented (see Haugeland 1978).

[2] As explained in AOS, the aim is to include not only the machines, devices, implementations, architectures, programs, processes, algorithms, languages, networks, interactions, behaviours, interfaces, etc., that constitute computing, but also the design, implementation, maintenance, and even use of such systems (such as Microsoft Word). Not, of course, that a theory will explain any *particular* architecture, language, etc. Rather, the point is that a foundational theory should explain *what an architecture is*, what constraints architectures must meet, etc.

[3] Indeed, I ultimately argue that that theory—trafficking in Turing machines, notions of "effective computability", and the like—fails as a theory of computing, in spite of its name and its popularity. It is simultaneously

all proposals that assume that computation can be *defined*. By my lights, that is, computer science should be viewed as an empirical endeavor.[4] An adequate theory must make a substantive empirical claim about what I call *computation in the wild*:[5] that eruptive body of practices, techniques, networks, machines, and behaviour that has so palpably revolutionised late twentieth- and early twenty-first-century life.

The second, "conceptual" criterion, that a theory own up to— and as far as possible repay— its intellectual debts, is in a way no more than standard theoretical hygiene. But it is important to highlight, in the computational case, for two intertwined reasons. First, it turns out that several candidate theories of computing (including the official "Theory of Computation" mentioned above), as well as many of the reigning but largely tacit ideas about computing held in surrounding disciplines, implicitly rely, without explanation, on such substantial, recalcitrant notions as interpretation,[6] representation, and semantics.[7] Second, which only makes matters worse, there is a widespread tendency in the surrounding intellectual terrain to point to computation as a possible *theory of those very recalcitrant notions*. Unless we ferret out all such de-

too broad, in applying to more things than computers, and too narrow, in that it fails to apply to some things that are computers. More seriously, what it is a theory of, is not *computing*. See §5.2.

[4] Methodological issues arise, owing to the fact that we (at least seem to) make up the evidence. Although this ultimately has metaphysical as well as methodological implications, it undermines the empirical character of computer science no more than it does in, say, sociology or linguistics.

[5] Adapted from Hutchins' *Cognition in the Wild* (1995).

[6] 'Interpretation' is a technical notion in computing; how it relates to the use of the term in ordinary language, or to what 'interpretation' is thought to signify in literary or critical discussions, is typical of the sort of question to be addressed in the full analysis.

[7] A notable example of such a far-from-innocent assumption is the widespread theoretical tendency to distinguish (i) an abstract and presumptively fundamental notion of "computation" from (ii) a concrete but derivative notion of a "computer"—the latter simply being taken to be any physical device able to carry out a computation. It turns out, on inspection, that this assumption builds in a residually dualist stance towards what is essentially the mind/body problem—a stance I eventually want to argue against, and at any rate not a thesis that should be built into a theory of computing as a presumptive but inexplicit premise.

pendencies, and lay them in plain view, we run at least two serious risks: (i) of endorsing accounts that are either based on, or give rise to, vicious conceptual circularity; and (ii) of promulgating and legitimating various unwarranted preconceptions or parochial (e.g., modernist) biases— such as of a strict mind-body dualism.

The third "cognitive" criterion—that an adequate theory of computation provide an intelligible foundation for a theory of mind—is of a somewhat different character. Like the second, it is more a metatheoretic requirement on the form of a theory than a constraint on its substantive content. But its elevation to a primary criterion is non-standard, and needs explaining.

Its inclusion is not simply based on the fact that the computational theory of mind (the idea that we, too, might be computers) is one of the most provocative and ramifying ideas in intellectual history, underwriting artificial intelligence, cognitive psychology, and contemporary philosophy of mind. Several other ideas about computing are just as sweeping in scope (such as proposals to unify the foundations of quantum mechanics with the foundations of information), but have not spawned their own methodological criteria. Rather, what distinguishes the computational theory of mind, in the present context, has to do with the epistemological consequences that would follow, if it were true.

Theorizing is undeniably a cognitive endeavor. If the computational theory of mind were correct, therefore, a theory of computation would be *reflexive*—applying not only (at the object-level) to computing in general, but also (at the meta-level) to the process of theorizing. That is, the theory's claims about the nature of computing would apply to the theory itself. On pain of contradiction, therefore, unless one determines the reflexive implications of any candidate theory (of computing) on the form that the theory itself should take, and assesses the theory from such a reflexively consistent position, one will not be able to judge whether it is correct.[8]

[8] For example, it would be inconsistent simultaneously to claim the following three things: (i) as many do, that scientific theories should be expressed from an entirely third-person, non-subjective point of view; (ii) as an intrinsic fact about all computational processes, that genuine reference is possible only from a first-person, subjective vantage point ("first-person" from the perspective of the machine, that is); and (iii) that the computa-

More specifically, suppose that mind is in fact computational, and that we were to judge a candidate (object-level) theory of computing from the perspective of an implicit meta-theory inconsistent with that candidate theory. And then suppose that, when judged from that perspective, the candidate theory is determined to be good or bad. There would be no reason to trust such a conclusion. For the conclusion might be due not to the empirical adequacy or failings of the theory under consideration, but rather to the conceptual inadequacy of the presumed meta-theory.[9]

In sum, the plausibility of the computational theory of mind requires that a proper analysis of a candidate theory of computing must consider: (i) what computational theory of mind would be generated, in its terms; (ii) what form theories in general would take, on such a model of mind; (iii) what the candidate theory of computing in question would look like, when framed as such a theory; (iv) whether the resulting theory (of computing), so framed, would hold true of computation-in-the-wild; and (v) whether, if it did turn out to be true (i.e., empirically), mentation and theorizing would, by those lights, also be computational. *All this is required, for reflexive integrity.* To do these things, we need to understand whether—and how—the theory could underwrite a theory of mind. Hence the cognitive criterion.

It is essential to understand, however, that the cognitive criterion requires only that we *understand* what form a computational theory of mind would take; it does not reflect any commitment to *accept* such a theory. In committing myself to honor the criterion, that is, I make no advance commitment to computationalism's being true or false. I just want to know what it says.

None of this is to say that the content of the computational theory of mind is left open. Computationalism's fundamental thesis—that the mind is computational—is given substance by the first, empirical criterion. Computationalism, that is—at least as I

tional theory of mind is true. If one were to believe in the ineliminably first-person character of computational reference, and that human reference is a species of computational reference, then consistency would demand that such a theory be stated *from a first-person point of view*—since, by hypothesis, no other way of presenting the theory would refer.

[9] Note that the situation is symmetric; reflexive inconsistencies can generate both false negatives and false positives.

read it—is not a theory-laden or "opaque" proposal, in the sense of framing or resting on a specific hypothesis about what computers are. Rather, it has more an ostensive or "transparent" character: it claims that people (i.e., us) are computers in whatever way that computers (i.e., those things over there) are computers, or at least in whatever way *some* of those things are computers.[10]

It follows that specific theoretical formulations of computationalism (whether pro or con) are doubly contingent. Thus consider, on the positive side, Newell and Simon's popular (1976) "physical symbol system hypothesis," according to which human intelligence is claimed to consist of physical symbol manipulation; or Fodor's (1975, 1980) claim that thinking consists of formal symbol manipulation; or Dreyfus' (1992) assertion that computationalism (as opposed to connectionism) requires the explicit manipulation of explicit symbols; or—on the critical side—van Gelder's (1996) claim that computationalism is both false and misleading, deserving to be replaced by dynamical alternatives. Not only do all these writers make hypothetical statements about *people*, that they are or are not physical, formal, or explicit symbol manipulators, respectively; they do so by making (hypothetical) statements about *computers*, that they are in some essential or illuminating way characterizable in the same way. Because I take the latter claims to be as subservient to empirical adequacy as the former, there are two ways in which these writers could be wrong. In claiming that people are formal symbol manipulators, for example, Fodor would naturally be wrong if computers were formal symbol manipulators and people were not. But he would also be wrong, *while the computational theory of mind itself might still be true*, if computers were not formal symbol manipulators, either. Similarly, van Gelder's brief against computational theories of mind is vulnerable to his understanding of what computing is actually like. If, as I believe, computation-in-the-wild is not as he characterises it, then the sting of his critique is entirely eliminated.

[10] The computational theory of mind does not claim that minds and computers are equivalent (in the sense that anything that is a mind is a computer, and vice versa). Rather, the idea is that minds are (at least) a *kind* of computer, and furthermore that the kind is *itself computationally characterised* (i.e., that the characteristic predicate on the restricted class of computers that are minds is itself to be framed in computational terms).

In sum, computational cognitive science is, like computer science, hostage to the foundational project:[11] of formulating a comprehensive, true, and intellectually satisfying theory of computing that honors all three criteria.

Not one of them is easy to meet.

3 Seven Construals of Computing

Some will argue that we already know what computation is. That in turn breaks into two questions: (i) is there a story—an account that people think answers the question of what computing is (what computers are); and (ii) is that story right?

Regarding the first question, the answer is not *no*, but it is not a simple *yes*, either. More than one idea is at play in current theoretic discourse. Over the years, I have found it convenient to distinguish seven primary *construals* of computation, each requiring its own analysis:

1. **Formal Symbol Manipulation (FSM):** the idea, derivative from a century's work in formal logic and metamathematics, of a machine manipulating symbolic or (at least potentially) meaningful expressions without regard to their interpretation or semantic content;

2. **Effective Computability (EC):** what can be done, and how hard it is to do it, mechanically, as it were, by an abstract analogue of a "mere machine";

3. **Execution of an algorithm (ALG) or rule-following (RF):** what is involved, and what behaviour is thereby produced,

[11] Foundationalism is widely decried, these days—especially in social and critical discourses. Attempting a foundational reconstruction of the sort I am attempting here may therefore be discredited, by some, in advance. As suggested in Smith (1996), however, I do not believe that any of the arguments that have been raised against foundationalism (particularly: against the valorization of a small set of types or categories as holding an unquestioned and/or uniquely privileged status) amounts to an argument against rigorously plumbing the depths of an intellectual subject matter. In this paper, my use of the term 'foundational' should be taken as informal and, to an extent, lay (I am as committed as anyone to the fallacies and even dangers of master narratives, ideological inscription, and/or uniquely privileging any category or type).

in following a set of rules or instructions, such as when making dessert;

4. **Calculation of a Function (FUN):** the behaviour, when given as input an argument to a mathematical function, of producing as output the value of that function applied to that argument;

5. **Digital State Machine (DSM):** the idea of an automaton with a finite, disjoint set of internally homogeneous machine states—as parodied in the "clunk, clunk, clunk" gait of a 1950's cartoon robot;

6. **Information Processing (IP):** what is involved in storing, manipulating, displaying, and otherwise trafficking in information, whatever information might be; and

7. **Physical Symbol Systems (PSS):** the idea, made famous by Newell and Simon (1976), that, somehow or other, computers interact with, and perhaps also are made of, symbols in a way that depends on their mutual physical embodiment.

These seven construals have formed the core of our thinking about computation over the last fifty years, but no claim is made that this list is exhaustive.[12] At least to date, however, it is these seven that have shouldered the lion's share of responsibility for framing the intellectual debate.

By far the most important step in getting to the heart of the foundational question, I believe, is to recognise that these seven construals are all conceptually distinct. In part because of their great familiarity (we have long since lost our innocence), and in part because "real" computers seem to exemplify more than one of them—including those often-imagined but seldom-seen Turing machines, complete with controllers, read-write heads, and indefinitely long tapes—it is sometimes uncritically thought that all seven can be viewed as rough synonyms, as if they were different ways of getting at the same thing. Indeed, this conflationary tendency is rampant in the literature, much of which moves around among them as if doing so were intellectually free. But that is a

[12] See the sidebar at the top of the next page.

mistake. The supposition that any two of these construals amount to the same thing, let alone that all seven do, is simply false.

For example, consider the formal symbol manipulation construal (FSM). It explicitly characterises computing in terms of a semantic or intentional aspect, if for no other reason than that without some such intentional character there would be no warrant in calling it *symbol* manipulation.[13] In contrast, the digital state machine construal (DSM) makes no such reference to intentional properties. If a Lincoln-log contraption were digital but not symbolic, and a system manipulating continuous symbols were formal but not digital, they would be differentially counted as computational by the two construals. Not only do FSM and DSM *mean* different things, in other words; they (at least plausibly) have overlapping but distinct extensions.

The effective computability (EC) and algorithm execution (ALG) construals similarly differ on the crucial issue of semantics. Whereas the effective computability construal, at least in the hands of computer scientists, seems free of intentional connotation,[14] the idea of algorithm execution, at least as I have characterised it, seems not only to involve rules or recipes, which presumably do mean something, but also (pace Wittgenstein) to require some sort of understanding on the part of the agent producing the behaviour.

Semantics is not the only open issue; there is also an issue of abstractness versus concreteness. For example, it is unclear whether the notions of "machine" and "taking an effective step" internal to the EC construal make fundamental reference to causal powers, material realization, or other concrete physical properties, or whether, as most current theoretical discussions suggest, effective computability should be taken as an entirely abstract mathematical notion. Again, if we do not understand this crucial aspect of the "mind-body problem for machines," how can we expect computational metaphors to help us in the case of people?

[13] See footnote 22.

[14] At least some logicians and philosophers, in contrast, do read the effective computability construal semantically. This difference is exactly the sort of question that needs to be disentangled and explained in the full analysis.

Additional Construals

Especially as the boundaries between computer science and surrounding intellectual territory erode (itself a development predicted by this analysis; see section 8), several ideas that originated in other fields are making their way into the center of computational theorizing as alternative conceptions of computing. At least three are important enough to be seen as construals in their own right (though the first is not usually assumed to have any direct connection with computing, and the latter two are not normally assumed to be quite as "low-level" or foundational as the primary seven):

8. **Dynamics (DYN):** the notion of a dynamical system, linear or non-linear, as popularized in discussions of attractors, turbulence, criticality, emergence, etc.;
9. **Interactive Agents (IA):** active agents enmeshed in an embedding environment, interacting and communicating with other agents (and perhaps also with people); and

There are still other differences among construals. They differ on whether they inherently focus on internal structure or external input/output, for example—that is, on whether: (i) they treat computation as fundamentally *a way of being structured or constituted*, so that surface or externally observable behaviour is derivative; or whether (ii) the *having of a particular behaviour* is the essential locus of being computational, with questions about how that is achieved left unspecified and uncared about. The formal symbol manipulation and digital state machine construals are of the former, structurally constitutional sort; effective computability is of the latter, behavioural variety; algorithm execution appears to lie somewhere in the middle.

The construals also differ in the degree of attention and allegiance they have garnered in different disciplines. Formal symbol manipulation (FSM) has for many years been the conception of computing that is privileged in artificial intelligence and philosophy of mind, but it receives almost no attention in computer science. Theoretical computer science focuses primarily on the effective computability (EC) and algorithm (ALG) construals, whereas mathematicians, logicians, and most philosophers of logic and mathematics pay primary allegiance to the functional conception (FUN). Publicly, in contrast, it is surely the information processing

10. **Self-organizing** or **Complex Adaptive Systems (CAS):** a notion—often associated with the Santa Fe Institute—of self-organizing systems that respond to their environment by adjusting their organization or structure, so as to survive and (perhaps even) prosper.

Additional construals may need to be added, over time. Moreover, there are even those who deny that computation has *any* ontologically distinct identity. Thus Agre (1997b), for example, claims that computation should instead be methodologically individuated (note that this eviscerates the computational theory of mind).

11. **Physical Implementation (PHY):** a methodological hypothesis that computation is not ontologically distinct, but rather that computational practice is human expertise in the physical or material implementation of (apparently arbitrary) systems.

(IP) construal that receives the major focus—being by far the most likely characterization of computation to appear in the *Wall Street Journal*, and the idea responsible for such popular slogans as "the information age" and "the information highway."

Not only must the seven construals be distinguished one from another; additional distinctions must be made within each one. Thus the idea of information processing (IP) needs to be broken down, in turn, into at least three sub-readings, depending on how 'information' is understood: (i) as a *lay* notion, dating from perhaps the nineteenth-century, of something like an abstract, publicly-accessible commodity, carrying a certain degree of autonomous authority; (ii) so-called "information theory," an at least seemingly semantics-free notion that originated with Shannon and Weaver (1949), spread out through much of cybernetics and communication theory, is implicated in Kolmogorov, Chaitin, and similar complexity measures, and has more recently been tied to notions of energy and, particularly, entropy; and (iii) the semantical notion of information advocated by Dretske (1981), Barwise and Perry (1983), Halpern (1987), and others, which in contrast to the second deals explicitly with semantic content and veridicality.

Clarifying all these issues, bringing the salient assumptions to the fore, showing where they agree and where they differ, tracing the roles they have played in the last fifty years—questions like this must be part of any foundational reconstruction. But in a sense these issues are all secondary. For none has the bite of the second question raised at the beginning of the section: of whether any of the enumerated accounts is *right*.

Naturally, one has to say just what this question means—has to answer the question "Right of what?"—in order to avoid the superficial response: "Of course such and such a construal is right; that's how computation is *defined!*" This is where the empirical criterion takes hold. More seriously, I am prepared to argue for a much more radical conclusion, which we can dub as the first major result:[15]

C1. When subjected to the empirical demands of practice and the (reflexively mandated) conceptual demands of cognitive science, *all seven primary construals fail*—for deep, overlapping, but distinct, reasons.

4 Diagnosis I: General

What is the problem? Why do these theories all fail?

The answers come at many levels. In the next section I discuss some construal-specific problems. But a general thing can be said first. Throughout, the most profound difficulties have to do with semantics. It is widely (if tacitly) recognised that computation is in one way or another a symbolic or representational or information-based or semantical—that is, as philosophers would say, an *intentional*—phenomenon.[16] Somehow or other, though in ways

[15] This numbering system (**C1**–**C9**) is used only for purposes of this paper; it will not necessarily be used in AOS.

[16] Although the term 'intentional' is primarily philosophical, there are many philosophers, to say nothing of some computer and cognitive scientists, who would deny that computation is an intentional phenomenon. Reasons vary, but the most common goes something like this: (i) that computation is both *syntactic* and *formal*, where 'formal' means "independent of semantics"; and (ii) that intentionality has fundamentally to do with semantics; and therefore (iii) that computation is thereby not intentional. I believe this is wrong, both empirically (that computation is purely syntactic) and conceptually (that being syntactic is a way of not being intentional); I also disagree that being intentional has *only* to do with semantics, which the denial requires. See footnote 22.

we do not yet understand, the states of a computer can model or simulate or represent or stand for or carry information about or signify other states in the world (or at least can be taken by people to do so). This semantical or intentional character of computation is betrayed by such phrases as *symbol* manipulation, *information* processing, programming *languages*, *knowledge representation*, *databases*, and so on. Indeed, if computing were not intentional, it would be spectacular that so many intentional words of English systematically serve as technical terms in computer science.[17] Furthermore—and this is important to understand—it is the intentionality of the computational that motivates the cognitivist hypothesis. The only compelling reason to suppose that we (or minds or intelligence) might be computers stems from the fact that we, too, deal with representations, symbols, meaning, information, and the like.[19]

For someone with cognitivist leanings, therefore—as opposed, say, to an eliminativist materialist, or to some types of connectionist—it is natural to expect that a comprehensive theory of computation will have to focus on its semantical aspects. This raises problems enough. Consider just the issue of representation. To meet the first criterion, of empirical adequacy, a successful candidate will have to make sense of the myriad kinds of representation that saturate real-world systems— from bit maps and images to knowledge representations and databases; from high-speed caches to long-term backup tapes; from low-level finite-element models used in simulation to high-level analytic descriptions supporting reasoning and inference; from text to graphics to audio to video to virtual reality. As well as being vast in scope, it will also have to combine decisive theoretical bite with exquisite resolution, in order to distinguish: models from implementations; analyses from simulations; and virtual machines at one level of abstraction from virtual machines at another level of abstraction, in terms of which the former may be implemented.

[17] Thus computer science's use of (the English words) 'language,' 'representation,' 'data,' etc. is analogous to physics' use of 'work,' 'force,' 'energy,' etc.—as opposed to its use of 'charm.' That is, it reflects a commitment to do scientific justice to the center of gravity of the word's natural meaning, rather than being mere whimsical fancy.

[18] Physically, we and (at least contemporary) computers are not very much

To meet the second, conceptual criterion, moreover, any account of this profusion of representational practice must be grounded on, or at least defined in terms of, a theory of semantics or content, partly in order for the concomitant psychological theory to avoid vacuity or circularity, and partly so that even the computational part of the theory meet a minimal kind of naturalistic criterion: that we understand how computation is part of the natural world. This is made all the more difficult by the fact that the word 'semantics' is used in an incredible variety of senses across the range of the intentional sciences. Indeed, in my experience it is virtually impossible, from any one location within that range, to understand the full significance of the term, so disparate is that practice *in toto*.[19]

Genuine theories of content, moreover—of what it is that makes a given symbol or structure or patch of the world be *about* or *oriented towards* some other entity or structure or patch—are notoriously hard to come by.[20] Some putatively foundational construals of computation are implicitly defined in terms of just such a background theory of semantics, but neither explain what semantics is, nor admit that semantical dependence—and thus fail the second, conceptual criterion. This includes the first, formal symbol manipulation construal so favored (and disparaged!) in the cognitive

alike—though it must be said that one of the appeals, to some people at least, of the self-organizing or complex-adaptive-system construal (CAS) is its prospect of providing a naturalistically palatable and non-intentional but nevertheless specific way of discriminating people-cum-computers (and perhaps higher animals) from arbitrary physical devices.

[19] In computer science, to take a salient example, the term "the semantics of α", where α is an expression or construct in a programming language, means approximately the following: the topological (as opposed to geometrical) temporal profile of the behaviour to which execution of this program fragment gives rise. By 'topological' I mean that the overall temporal order of events is dictated, but that their absolute or metric time-structure (e.g., exactly how fast the program runs) is not. As a result, a program can usually be sped up, either by adjusting the code or running it on a faster processor, without, as is said, "changing the semantics."

[20] Best known are Dretske's semantic theory of information (1981), which has more generally given rise to what is known as "indicator semantics"; Fodor's "asymmetrical-dependence" theory (1987); and Millikan's "teleosemantics" or "biosemantics" (1984, 1989). For comparison among these alternatives see, e.g., Fodor (1984) and Millikan (1990).

sciences, in spite of its superficial formulation as being "independent of semantics."[21] Other construals, such as those that view computation as the behaviour of discrete automata—and also, I will argue below, even if this is far from immediately evident, the recursion-theoretic one that describes such behaviour as the calculation of effective functions—fail to deal with computation's semantical aspect at all, in spite of sometimes using semantical vocabulary, and so fail the first, empirical criterion. In the end, one is inexorably driven to a second major conclusion:

C2. In spite of the advance press, especially from cognitivist quarters, computer science, far from supplying the answers to fundamental intentional mysteries, must, like cognitive science, await the development of a satisfying theory of semantics and intentionality.[22]

[21] Because formal symbol manipulation is usually defined as "manipulation of symbols independent of their interpretation", some people believe that the formal symbol manipulation construal of computation does not rest on a theory of semantics. But that is simply an elementary, though apparently common, conceptual mistake. As discussed further in section 5, the "independence of semantics" postulated as essential to the formal symbol construal is independence at the level of the phenomenon; it is a claim about how symbol manipulation *works*. Or so at least I believe, based on many years of investigating what practitioners are actually committed to (whether it is *true*—i.e., holds of computation-in-the-wild—is a separate issue). The intuition is simple enough: that semantic properties, such as referring to the Sphinx, or being true, are not of the right sort to do effective work—so they cannot be the sort of property in virtue of the manifestation of which computers *run*. At issue in the present discussion, in contrast, is a more logical form of independence, *at the level of the theory* (or, perhaps, to put it more ontologically and less epistemically, independence at the level of the *types*). Here the formal symbol manipulation construal is as dependent on semantics as it is possible to be: *it is defined in terms of it*. And (as the parent of any teenager knows) defining yourself in opposition to something is not ultimately a successful way of achieving independence. Symbols must have a semantics, in other words (have an actual interpretation, or be interpretable, or whatever), in order for there to be something substantive for their formal manipulation to proceed independently of. Without a semantic character to be kept crucially in the wings, the formal symbol manipulation construal would collapse in vacuity—would degenerate into something like "the manipulation of structure" or, as I put it in *AOS*, "stuff manipulation"—i.e., materialism.

[22] As suggested in the preceding footnote, philosophers are less likely than

5 Diagnosis II: Specific

So none of the seven construals provides an account of semantics. Since I take computation to be semantic (not just by assumption, but as an unavoidable lesson from empirical investigation), that means they fail as theories of computation, as well (i.e., **C2** implies **C1**). And that is just the beginning of the problems. All seven also fail for detailed structural reasons—different reasons per construal, but reasons that add up, overall, to a remarkably coherent overall picture.

In this section I summarise just a few of the problems, to convey a flavor of what is going on. In each case, to put this in context, these results emerge from a general effort, in the main investigation, to explicate, for each construal:

1. What the construal says or comes to—what claim it makes about what it is to be a computer;
2. Where it derives from, historically;
3. Why it has been held;
4. What's right about it—what insights it gets at;
5. What is wrong with it, conceptually, empirically, and explanatorily;
6. Why it must ultimately be replaced; and
7. What about it should nevertheless be retained in a "successor," more adequate account.

5a Formal Symbol Manipulation

The FSM construal has a distinctly *antisemantical* flavor, owing to its claim that computation is the "manipulation of symbols independent of their semantics." On analysis, it turns out to be motivated by two entirely different, ultimately incompatible, independ-

computer scientists to expect a theory of computation to be, or to supply, a theory of intentionality. That is, they would not expect the metatheoretic structure to be as expected by most computer scientists and artificial intelligence researchers—namely, to have a theory of intentionality *rest on* a theory of computation. But that does not mean they would necessarily agree with the opposite, which I am arguing here: that a theory of computation will have to rest on a theory of intentionality. Many philosophers seem to think that a theory of computation can be *independently* of a theory of intentionality. Clearly, I do not believe this is correct.

ence intuitions. The first motivation is at the level of the theory, and is reminiscent of a reductionist desire for a "semantics-free" account. It takes the FSM thesis to be a claim that computation can be *described* or *analysed* in a semantics-free way. If that were true, so the argument goes, that would go some distance towards naturalizing intentionality.[23]

There is a second motivating intuition, different in character, that holds at the level of the phenomenon. Here the idea is simply the familiar observation that intentional phenomena, such as reasoning, hoping, or dreaming, carry on in relative independence of their subject matters or referents. Reference and truth, it is recognised, are just not the sorts of properties that can play a causal role in engendering behaviour—essentially because they involve some sort of relational coordination with things that are *too far away* (in some relevant respect) to make a difference. This relational characteristic of intentionality—something I call semantic *disconnection*—is such a deep aspect of intentional phenomena that it is hard to imagine its being false. Without it, falsity would cease to exist, but so too would hypotheticals; fantasy lives would be metaphysically banned; you would not be able to think about continental drift without bringing the tectonic plates along with you.

For discussion, I label the two readings of the formal symbol manipulation construal *conceptual* and *ontological*, respectively.[24] The ontological reading is natural, familiar, and based on a deep insight. But it is too narrow. Many counterexamples can be cited against it, though space does not permit rehearsing them here.[25] Instead, to get to the heart of the matter, it helps to highlight a distinction between two kinds of "boundary" thought to be relevant or essential—indeed, often assumed *a priori*—in the analysis of computers and other intentional systems:

1. **Physical:** A physical boundary between the system and its surrounding environment—between "inside" and "out-

[23] As Haugeland says "... if you take care of the syntax, the semantics will take care of itself" (1981a, 23); see also Haugeland (1985).

[24] It can be tempting to think of the two readings as corresponding to intensional and extensional readings of the phrase "independent of semantics"—but that isn't strictly correct. See *AOS*.

[25] See *AOS Volume II*.

side"; and

2. **Semantic:** A semantic "boundary" between symbols and their referents.

In terms of these two distinctions, the ontological reading of the FSM construal can be understood as presuming the following two theses:

1. **Alignment:** That the physical and semantic boundaries line up, with all the symbols inside, all the referents outside; and

2. **Isolation:** That this allegedly aligned boundary is a barrier or gulf across which various forms of dependence (causal, logical, explanatory) do not reach.

The fundamental idea underlying the FSM thesis, that is, is that a barrier of this double allegedly-aligned sort can be drawn around a computer, separating a pristine inner world of symbols—a private kingdom of ratiocination or thought, as it were—understood both to work (ontologically) and to be analyzable (theoretically) in isolation, without distracting influence from the messy, unpredictable exterior.

It turns out, in a way that is ultimately not surprising, that the traditional examples motivating the FSM construal, such as theorem proving in formal logic, meet this complex pair of conditions. First, they involve internal symbols designating external situations, thereby satisfying ALIGNMENT (internal) databases representing (external) employee salaries, (internal) differential equations modeling the (external) perihelion of Mercury, (internal) first-order axioms designating (external) Platonic numbers or purely abstract sets, and so on. Second, especially in the paradigmatic examples of formal axiomatizations of arithmetic and proof systems of first-order logic (and, even more especially, when those systems are understood in classical, especially model-theoretic, guise), the system is assumed to exhibit the requisite lack of interaction between the (internal) syntactic proof system and the (external, perhaps model-theoretic) interpretation, satisfying ISOLATION. In conjunction, the two assumptions allow the familiar two-part picture of a formal system to be held: a locally contained syntactic system, on the one hand, consisting of symbols or formulae in close causal intimacy with a proof-theoretic inference regimen; and a remote realm of numbers or sets or "ur-elements," in which the symbols or formulae are interpreted, on the other. It is because the formality condition relies on both theses together that the classical picture takes computation to consist exclusively of symbol-symbol transformations, carried on entirely within the confines of a machine.

The first—and easier—challenge to the antisemantical thesis comes when one retains the first ALIGNMENT assumption, of coincident boundaries, but relaxes the second ISOLATION claim, of no interaction. This is the classical realm of input/ output, home of the familiar notion of a transducer. And it is here that one encounters the most familiar challenges to the FSM construal, such as the "robotic" and "system" replies to Searle's (1980) Chinese room argument, and Harnad's (1990) "Total Turing Test" as a measure of intelligence. Thus imagine a traditional perception system—for example, one that on encounter with a mountain lion constructs a symbolic representation of the form MOUNTAIN-LION-043. There is interaction (and dependence) from external world to internal representation. By the same token, an actuator system, such as one that would allow a robot to respond to a symbol of the form CROSS-THE-STREET by moving from one side of the road to the other, violates the independence assumption in the other direction, from internal representation to external world.

Note, in spite of this interaction, and the consequent violation of ISOLATION, that ALIGNMENT is preserved in both cases: the transducer is imagined to mediate between an internal symbol and an external referent. Nevertheless, the violation of ISOLATION is already enough to defeat the formality condition. This is why transducers and computation are widely recognised to be uneasy bedfellows, at least when formality is at issue. It is also why, if one rests the critique at this point, defenders of the antisemantical construal are tempted to wonder, given that the operations of transducers violate *formality*, whether they should perhaps be counted as *not being computational*.[26] Given the increasing role of environ-

[26] Thus Devitt (1991) restricts the computational thesis to what he calls "thought-thought" (t-t) transactions; for him output (t–o) and input (i–t)

mental interaction within computational practice, it is not at all clear that this would be possible, without violating the condition of empirical adequacy embraced at the outset. For our purposes it doesn't ultimately matter, however, because the critique is only halfway done.

More devastating to the FSM construal are examples that challenge the ALIGNMENT thesis. It turns out, on analysis, that far from lining up on top of each other, real-world computer systems' physical and semantic boundaries *cross-cut*, in rich and productive interplay. It is not just that computers are involved in an engaged, participatory way with *external* subject matters, in other words, as suggested by some recent "situated" theorists. They are participatorily engaged in the world *as a whole*—in a world that indiscriminately includes themselves, their own internal states and processes. This integrated participatory involvement, blind to any *a priori* subject-world distinction, and concomitantly intentionally directed towards both internally and externally exemplified states of affairs, is not only architecturally essential, but is also critical, when the time comes, in establishing and grounding a system's intentional capacities.

From a purely structural point of view, four types of case are required to demonstrate this non-alignment of boundaries: (i) where a symbol and referent are both internal; (ii) where a symbol is internal and its referent external; (iii) where symbol and referent are both external; and (iv) where symbol is external and referent internal. The first is exemplified in cases of quotation, metastructural designation, window systems, e-mail, compilers, loaders, network routers, and at least arguably all programs (as opposed, say, to databases). The second, of internal symbols with external referents, can be considered as something of a theoretical (though not necessarily empirical) default, as for example when one reflects on the sun's setting over Georgian Bay (to use a human example), or when a computer database represents the usage pattern of a set of university classrooms. The third and fourth are neither more nor less than a description of ordinary written text, public writing, etc.—to say nothing of pictures, sketches, conversations, and the whole panoply of other forms of external representa-

transactions count as non-computational.

tion. Relative to any particular system, they are distinguished by whether the subject matters of those external representations are similarly external, or are internal. The familiar red skull-and-cross-bones signifying radioactivity is external to both man and machine, and also denotes something external to man and machine, and thus belongs to the third category. To a computer or person involved, on the other hand, an account of how they work (psychoanalysis of person or machine, as it were, to say nothing of logic diagrams, instruction manuals, etc.) is an example of the fourth.

By itself, violating ALIGNMENT is not enough to defeat formality. What it does accomplish, however, is to radically undermine ISOLATION's plausibility. In particular, the antisemantical thesis constitutive of the FSM construal is challenged not only because these examples show that the physical and semantic boundaries cross-cut, thereby undermining the ALIGNMENT assumption, but because they illustrate the presence, indeed the prevalence, of effective traffic across both boundaries—between and among all the various categories in question—thereby negating ISOLATION.

And this negation of ISOLATION, in turn, shows up, for what it is, the common suggestion that transducers, because of violating the antisemantical thesis, should be ruled "out of court"— i.e., should be taken as non-computational, à la Devitt (1991).[27] It should be clear that this maneuver is ill-advised; even a bit of a cop-out. For consider what a proponent of such a move must face up to, when confronted with boundary non-alignment. *The notion of a transducer must be split in two.* In order to retain an antisemantical (FSM) construal of computing, someone interested in transducers would have to distinguish:

1. **Physical transducers,** for operations or modules that cross or mediate between the inside and outside of a system; and

2. **Semantic transducers,** for operations or modules that mediate or "cross" between symbols and their referents.

And it is this bifurcation, finally, that irrevocably defeats the antisemantical formalists' claim. For the only remotely plausible no-

[27] See the preceding footnote.

tion of transducer, in practice, is the physical one. That is what we think of when we imagine vision, touch, smell, articulation, wheels, muscles, and the like: systems that mediate between the internals of a system and the "outside" world. Transducers, that is, at least in informal imagination of practitioners, are for connecting systems to their (physical) environments.[28] What poses a challenge to the formal (antisemantical) symbol manipulation construal of computation, on the other hand, are *semantic* transducers: those aspects of a system that involve trading between occurrent states of affairs, on the one hand, and representations of them, on the other. Antisemantics is challenged as much by disquotation as by driving around.

As a result, the only way to retain the ontological version of the FSM construal is to disallow (i.e., count as non-computational) the operations of semantic transducers. *But that is absurd!* It makes it clear, ultimately, that distinguishing that subset of computation which satisfies the ontological version of the antisemantical claim is not only unmotivated, solving the problem by fiat (making it uninteresting), but is a spectacularly infeasible way to draw and quarter any actual, real-life system. For no one who has ever built a computational system has ever found any reason to bracket reference-crossing operations, or to treat them as a distinct type. Not only that; think of how many different kinds of examples of semantic transducer one can imagine: counting, array indexing, e-mail, disquotation, error-correction circuits, linkers, loaders, simple instructions, database access routines, pointers, reflection principles in logic, index operations into matrices, most Lisp primitives, and the like. Furthermore, to *define* a species of transducer in this semantical way, and then to remove them from consideration as not being genuinely computational, would make computation (minus the transducers) antisemantical *tautologically*. It would no longer be an interesting claim on the world that computation was antisemantical—an insight into how things are. Instead, the word 'computation' would simply be shorthand for antisemanti-

[28] This statement must be understood within the context of computer science, cognitive science, and the philosophy of mind. It is telling that the term 'transducer' is used completely differently in engineering and biology (its natural home), to signify mechanisms that mediate changes in *medium*, not that cross *either* the inside/outside *or* the symbol/referent boundary.

cal symbol manipulation. The question would be whether anything interesting was in this named class—and, in particular, whether this conception of computation captured the essential regularities underlying practice. And we have already seen the answer to that: it is *no*.

In sum, introducing a notion of a semantical transducer solves the problem tautologically, cuts the subject matter at an unnatural joint, and fails to reconstruct practice. That is quite a lot to have going against it.

Furthermore, to up the ante on the whole investigation, not only are these cases of "semantic transduction" all perfectly well-behaved; they even seem, intuitively, to be as "formal" as any other kind of operation. If that is so, then those systems either are not formal, after all, *or else the word 'formal' has never meant independence of syntax and semantics in the way that the* FSM *construal claims*. Either way, the ontological construal does not survive.

Though it has been framed negatively, we can summarise this result in positive terms:

C3. Rather than consisting of an internal world of symbols separated from an external realm of referents, as imagined in the FSM construal, real-world computational processes are *participatory*, in the following sense: they involve complex paths of causal interaction between and among symbols and referents, both internal and external, cross-coupled in complex configurations.

5b Effective Computability

Although different in detail, the arguments against the other major construals have a certain similarity in style. In each case, the strategy in the main investigation has been to develop a staged series of counterexamples, not simply to show that the construal is false, but to serve as strong enough intuition pumps on which to base a positive alternative. In other words, the point is not critique, but deconstruction en route to reconstruction. Space permits a few words about just one other construal: effective computability—the idea that underwrites recursion theory, complexity theory, and, as I have said, the official (mathematical) "Theory of Computation."

Note, for starters—as mentioned earlier—that whereas the first, FSM construal is predominant in artificial intelligence, cognitive science, and philosophy of mind, it is the second, effective computability (EC) construal, in contrast, that underlies most theoretical and practical computer science.

Fundamentally, it is widely agreed, the theory of effective computability focuses on "what can be done by a mechanism." But two conceptual problems have clouded its proper appreciation. First, in spite of its subject matter, it is almost always characterised *abstractly*, as if it were a branch of mathematics. Second, it is imagined to be a theory defined over (for example) the numbers. Specifically, the marks on the tape of the paradigmatic Turing machine are viewed as *representations*— representations, in general, or at least in the first instance, of numbers, functions, or other Turing machines.

In almost exact contrast to the received view, I argue two things. First, I claim that the theory of effective computability is fundamentally a theory about the *physical* nature of patches of the world. In underlying character, I believe, it is no more "mathematical" than anything else in physics— even if we use mathematical structures to model that physical reality. Second—and this is sure to be contentious—I argue that recursion theory is fundamentally a *theory of marks*. More specifically, rather than taking the marks on the tape to be representations of numbers, as has universally been assumed in the theoretical tradition, I defend the following claim:

C4. The representation relation for Turing machines, alleged to run from marks to numbers, in fact runs the other way, from numbers to marks. The truth is 180° off what we have all been led to believe.

In the detailed analysis various kinds of evidence are cited in defense of this non-standard claim. For example:

1. Unless one understands it this way, one can solve the halting problem;[29]
2. An analysis of history, through Turing's paper and subsequent work, especially including the development of the

[29] See *AOS: Volume III.*

universal Turing machine, shows how and why the representation relation was inadvertently turned upside down in this way;

3. The analysis makes sense of a number of otherwise-inexplicable practices, including, among other examples: (i) the use of the word "semantics" in practicing computer science to signify the behaviour engendered by running a program,[30] (ii) the rising popularity of such conceptual tools as Girard's linear logic, and (iii) the close association between theoretical computer science and constructive mathematics.

It follows from this analysis that all use of semantical vocabulary in the "official" Theory of Computation is metatheoretic. As a result, *the so-called (mathematical) "Theory of Computation" is not a theory of intentional phenomena*—in the sense that it is not a theory that deals with its subject matter *as* an intentional phenomena.

In this way the layers of irony multiply. Whereas the FSM construal fails to meet its own criterion, of being "defined independent of semantics," this second construal *does* meet (at least the conceptual reading of) that first-construal condition. Exactly in achieving that success, however, the recursion-theoretic tradition thereby fails. For computation, as was said above, and as I am prepared to argue, *is* (empirically) an intentional phenomenon. So the EC construal achieves naturalistic palatability at the expense of being about the wrong subject matter.

We are thus led inexorably to the following very strong conclusion: what goes by the name "Theory of Computation" fails not because it makes false claims about computation, but because *it is not a theory of computation at all.*[31, 32]

[30] See footnote 20.

[31] The fact that it is not a theory of computing does not entail that it does not *apply* to computers, of course. All it means is that, in that application, it is not a theory of them *as computers*.

[32] That the so-called theory of computation fails as a theory of computation because it does not deal with computation's intentionality is a result that should be agreed even by someone (e.g., Searle) who believes that computation's intentionality is inherently derivative. I myself do not believe that computation's intentionality is inherently derivative, as it happens, but even those who think it is must admit that it is still an intentional phenomenon

In sum, the longer analysis ultimately leads to a recommendation that we redraw a substantial portion of our intellectual map. What has been (indeed, by most people still is) called a "Theory of Computation" is in fact a general theory of the physical world—specifically, a theory of how hard it is, and what is required, for patches of the world in one physical configuration to change into another physical configuration. It applies to *all* physical entities, not just to computers. It is no more mathematical than the rest of physics, in using (abstract) mathematical structures to model (concrete) physical phenomena. Ultimately, therefore, it should be joined with physics—because in a sense it *is* physics.

We can put this result more positively. Though falsely (and misleadingly) labeled, the mathematical Theory of Computation has been a spectacular achievement, of which the twentieth-century should be proud. Indeed, this is important enough that we can label it as the fifth major result:

C5. Though not yet so recognised, the mathematical theory based on recursion theory, Turing machines, complexity analyses, and the like—widely known as the "Theory of Computation"—is neither more nor less than a *mathematical theory of causality.*

6 Method

Similarly strong conclusions can be arrived at by pursuing each of the other construals. Indeed, the conclusion from the analysis of the digital state machine construal (DSM)—that computation-in-the-wild is not digital—is, if anything, even more consequential than the results derived from either the FSM or the EC critiques. Rather than go into more construals here, however, I instead want to say a word about method—specifically, about *formality*. For a potent theme underlies all seven critiques: that part of what has blinded us to the true nature of computation has to do with the often pretheoretic assumption that *computation and/or computers are formal.*

In one way or another, no matter what construal they pledge al-

of some sort. For *derivative* does not mean *fake* or *false*. If "derivatively intentional" is not taken to be a substantive constraint, then we are owed (e.g., by Searle) an account of what *does* characterise computation.

legiance to, just about everyone thinks that computers are formal—that they manipulate symbols formally, that programs (formally) specify formal procedures, that data structures are a kind of formalism, that computational phenomena are uniquely suited for analysis by formal methods—and so on. In fact the computer is often viewed as the crowning achievement of an entire "formal tradition"—an intellectual orientation, reaching back through Galileo to Plato, that was epitomised in the twentieth century in the logic and metamathematics of Frege, Russell, Whitehead, Carnap, and Turing, among others.

This history would suggest that formality is an essential aspect of computation. But since the outset, I have not believed that this is necessarily right. For one thing, it has never been clear what the allegiance to formality is an allegiance to. It is not as if "formal" is a technical or theory-internal predicate, after all. People may believe that developing an idea means formalizing it, and that programming languages are formal languages, and that theorem provers operate on formal axioms— but few write "FORMAL(X)" in their daily equations. Moreover, a raft of different meanings and connotations of this problematic term lies just below the surface. Far from hurting, this apparent ambiguity has helped to cement popular consensus. Freed of the need to be strictly defined ('*formal*' is not a formal predicate), formality has been able to serve as a lightning rod for a cluster of ontological assumptions, methodological commitments, and social and historical biases.

Because it remains tacit, cuts deep, has important historical roots, and permeates practice, formality has been an ideal foil, over the years, with which to investigate computation.

Almost a dozen different readings of "formal" can be gleaned from informal usage: *precise*, *abstract*, *syntactic*, *mathematical*, *explicit*, *digital*, *a-contextual*, *non-semantic*, among others.[33] They are alike in foisting recalcitrant theoretical issues onto center stage.

[33] At one stage I asked a large number of people what they thought "formal" meant—not just computer scientists, but also mathematicians, physicists, sociologists, etc. It was clear from the replies that the term has very different connotations in different fields. Some mathematicians and logicians, for example, take it to be pejorative, in contrast to the majority of theoretical computer scientists, for whom it has an almost diametrically opposed positive connotation.

Consider explicitness, for example, of the sort that might explain such a sentence as “for theoretical purposes we should lay out our tacit assumptions in a formal representation.” Not only have implicitness and explicitness stubbornly resisted theoretical analysis, but both notions are parasitic on something else we do not understand: general representation.[34] Or consider “a-contextual.” Where is an overall theory of context in terms of which to understand what it would be to say of something (a logical representation, say) that it was not contextually dependent?

Considerations like this suggest that particular readings of formality can be most helpfully pursued within the context of the general theoretical edifices that have been constructed (more or less explicitly) in their terms. Five are particularly important:

1. The *antisemantical* reading mentioned above: the idea that a symbolic structure (representation, language, program, symbol system, etc.) is formal just in case it is manipulated *independent of its semantics*. Paradigmatic cases include so-called formal logic, in which it is assumed that a theorem—such as MORTAL(SOCRATES)— is derived by an automatic inference regimen without regard to the reference, truth, or even meaning of any of its premises.

2. A closely allied grammatical or *syntactic* reading, illustrated in such a sentence as “inference rules are defined in terms of the *formal* properties of expressions.” (Note that whereas the antisemantical reading is negatively characterised, this syntactic one has a positive sense.)

3. A reading meaning something like *determinate* or *well-defined*—that is, as ruling out all ambiguity and vagueness. This construal turns out to be related to a variant of the computationally familiar notion of digitality or discreteness.

[34] On its own, an eggplant cannot be either formal or explicit, at least not in its ordinary culinary role, since in that role it is not a representation at all. In fact the only way to make sense of calling something non-representational explicit is as short-hand for saying that it is explicitly represented (e.g., calling eggplant an explicit ingredient of moussaka as a way of saying that the recipe for moussaka mentions eggplant explicitly).

4. A construal of "formal" as essentially equivalent to *mathematical*.
5. A reading that cross-cuts the other four: formality as applied to analyses or *methods*, perhaps with a derivative ontological implication that some subject matters (including computation?) are uniquely suited to such analytic techniques.

The first two (antisemantical and syntactic) are often treated as conceptually equivalent, but to do that is to assume that a system's syntactic and semantic properties are *necessarily disjoint*—which is almost certainly false. The relationship between the third (determinate) reading and digitality does not have so much to do with what Haugeland (1982) calls "first-order digitality": the ordinary assumption that a system's states can be partitioned into a determinate set, such as that its future behaviour or essence stems solely from membership in one element of that set, without any ambiguity or matter of degree. Rather, vagueness and indefiniteness (as opposed to simple continuity) are excluded by a *second-order* form of digitality—digitality at the level of concepts or types, in the sense of there being a binary "yes/no" fact of the matter about whether any given situation falls under (or is correctly classified in terms of) the given concept. And finally, the fourth view—that to be formal has something to do with being mathematical, or at least with being mathematically characterizable—occupies something of an ontological middle-realm between the subject-matter orientation of the first three and the methodological orientation of the fifth.

The ultimate moral for computer and cognitive science, I argue, is similar to the claim made earlier about the seven construals: *not one of these readings of 'formal' correctly applies to the computational case*. It can never be absolutely proved that computation is not formal, of course, given that the notion of formality is not determinately tied down. What I am prepared to argue (and do argue in the full analysis) is the following: no standard construal of formality, including any of those enumerated above, is both (i) substantive and (ii) true of extant computational practice. Some readings reduce to vacuity, or to no more than physical realizability; others break down in internal contradiction; others survive the test of being substantial, but are demonstrably false of current systems. In the end, one is forced to a sixth major conclusion:

C6. Computation is not formal.

It is an incredible historical irony: the computer, darling child of the formal tradition, has outstripped the bounds of the very tradition that gave rise to it.

7 The Ontological Wall

Where does all this leave us? It begins to change the character of the project. It is perhaps best described in personal terms. Over time, investigations of the sort described above, and consideration of the conclusions reached through them, convinced me that none of the reigning theories or construals of computation, nor any of the reigning methodological attitudes towards computation, will *ever* lead to an analysis strong enough to meet the three criteria laid down at the outset.

It wasn't always that way. For the first twenty years of the investigation I remained:

1. In awe of the depth, texture, scope, pluck, and impact of computational practice;
2. Critical of the inadequate state of the current theoretical art;
3. Convinced that a formal methodological stance stood in the way of getting to the heart of the computational question; and
4. Sure in my belief that what was needed, above all else, was a *non-formal*—i.e., situated, embodied, embedded, indexical, critical, reflexive, all sorts of other things (it changed, over the years)—theory of representation and semantics, in terms of which to reconstruct an adequate conception of computing.

In line with this metatheoretic attitude, as the discussion this far will have suggested, I kept semantical and representational issues in primary theoretical focus. Since, as indicated in the last section, the official "Theory of Computation," derived from recursion and complexity theory, pays no attention to such intentional problems, to strike even this much of a semantical stance was to part company with the center of gravity of the received theoretical tradition.

You might think that this would be conclusion enough. And yet, in spite of the importance and magnitude of these intentional difficulties, and in spite of the detailed conclusions suggested above, I have gradually come to believe something much more sobering: a conclusion that, although not as precisely stated as the foregoing, is if anything even more consequential:

C7. The most serious problems standing in the way of developing an adequate theory of computation are as much *ontological* as *semantical*.

It is not that computation's semantic problems go away; they remain as challenging as ever. It is just that they are joined—on center stage, as it were— by even more demanding problems of ontology.

Except that to say "joined" is misleading, as if it were a matter of simple addition—as if now there were two problems on the table, whereas before there had been just one. No such luck. The two issues (representation and ontology) are inextricably entangled—a fact of obstinate theoretical and metatheoretical consequence.

A methodological consequence will illustrate the problem. Especially within the analytic tradition (by which I mean to include not just analytic philosophy, e.g., of language and mind, but most of modern science as well, complete with its formal/mathematical methods), it is traditional to analyse semantical or intentional systems, such as computers or people, under the following presupposition: (i) that one can parse or register the relevant theoretical situation in advance into a set of objects, properties, types, relations, equivalence classes, and so on (e.g., into people, heads, sentences, data structures, real-world referents, etc.)—as if this were theoretically innocuous—and then (ii), with that ontological parse in hand, go on to proclaim this or that or the other thing as an empirically justified result. Thus for example one might describe a mail-delivering robot by first describing an environment of offices, hallways, people, staircases, litter, and the like, through which the robot is supposed to navigate, and then, taking this characterization of its context as given, ask how or whether the creature represents routes, say, or offices, or the location of mail delivery stations.

If one adopts a reflexively critical point of view, however, as I have systematically been led to do (and as is mandated by the cognitive criterion), one is led inexorably to the following conclusion: that, in that allegedly innocent pretheoretical "set-up" stage, one is liable, even if unwittingly, to project so many presuppositions, biases, and advance "clues" about the "answer," and in general to so thoroughly prefigure the target situation, without either apparent or genuine justification, that *one cannot, or at least should not, take any of the subsequent "analysis" terribly seriously*. It is a general problem that I have elsewhere labelled *preemptive registration*.[35] It is problematic not just because it rejects standard analyses, but because it seems to shut all inquiry down. What else can one do, after all? How can one not parse the situation in advance (since it will hardly do to merely whistle and walk away)? And if, undaunted, one were to go ahead and parse it anyway, what kind of story could possibly serve as a justification? It seems that any conceivable form of defense would devolve into another instance of the same problem.

In sum, the experience is less one of facing an ontological challenge than of running up against a seemingly insuperable ontological wall. Perhaps not of slamming into it, at least in my own case; recognition dawned slowly. But neither is the encounter exactly gentle. It is difficult to exaggerate the sense of frustration that can come, once the conceptual fog begins to clear, from seeing one's theoretical progress blocked by what seems for all the world to be an insurmountable metaphysical obstacle.

Like many of the prior claims I have made, such as that all extant theories of computation are inadequate to reconstruct practice, or that no adequate conception of computing is formal, this last claim, that theoretical progress is stymied for lack of an adequate theory of ontology, is a strong statement, in need of correspondingly strong defense. Providing that defense is one of the main goals of *AOS*. In my judgment, to make it perfectly plain, despite the progress that has been made so far, and despite the recommended adjustments reached in the course of the seven specific

[35] Smith (in press).

analyses enumerated above, we are not going to get to the heart of computation, representation, cognition, information, semantics, or intentionality, until the ontological wall is scaled, penetrated, dismantled, or in some other way defused.

One reaction to the wall might be depression. Fortunately, however, the prospects are not so bleak. For starters, there is some solace in company. It is perfectly evident, once one raises one's head from the specifically computational situation and looks around, that computer scientists, cognitive scientists, and artificial intelligence researchers are not the only ones running up against severe ontological challenges. Similar conclusions are being reported from many other quarters. The words are different, and the perspectives complementary, but the underlying phenomena are the same.

Perhaps the most obvious fellow travelers are literary critics, anthropologists, and other social theorists, vexed by what analytic categories to use in understanding people or cultures that, by such writers' own admission, comprehend and constitute the world using concepts alien to the theorists' own. What makes the problem particularly obvious, in these cases, is the potential for *conceptual clash* between theorist's and subject's world view—a clash that can easily seem paralyzing. One's own categories are hard to justify, and reek of imperialism; it is at best presumptuous, and at worst impossible, to try to adopt the categories of one's subjects; and it is manifestly impossible to work with no concepts at all. So it is unclear how, or even whether, to proceed.

But conceptual clash, at least outright conceptual clash, is not the only form in which the ontological problem presents itself. Consider the burgeoning interest in self-organizing and complex systems mentioned earlier, currently coalescing in a somewhat renegade subdiscipline at the intersection of dynamics, theoretical biology, and artificial life. This community debates the "emergence of organization," the units on which selection operates, the structure of self-organizing systems, the smoothness or roughness of fitness landscapes, and the like. In spite of being disciplinarily constituting, however, these discussions are conducted in the absence of adequate theories of what organization is, of what a "unit" consist in, of how "entities" arise (as opposed to how they survive), of how it is determined what predicates should figure in characterizing a fitness landscape as rough or smooth, and so on. The ontological lack is to some extent recognised in increasingly vocal calls for "theories of organization."[36] But the calls have not yet been answered.

Ontological problems have also plagued physics for years, at least since foundational issues of interpretation were thrown into relief by the developments of relativity and quantum mechanics (including the perplexing wave-particle duality, and the distinction between "classical" and "quantum" world-views). They face connectionist psychologists, who, proud of having developed architectures that do not rely on the manipulation of formal symbol structures encoding high-level concepts, and thus of having thereby rejected propositional content, are nevertheless at a loss as to say what their architectures *do* represent. And then of course there are communities that tackle ontological questions directly: not just philosophy, but fields as far-flung as poetry and art, where attempts to get in, around, and under objects have been pursued for centuries.

So there are fellow-travelers. But no one, so far as I know, has developed an alternative ontological/metaphysical proposal in sufficient detail and depth to serve as a practicable foundational for a revitalised scientific practice. Unlike some arguments for realism or irrealism, unlike some briefs *pro* or *con* this or that philosophy of science, and unlike as well the deliberations of science studies and other anthropological and sociological and historical treatises about science, the task I have in mind is not the increasingly common *meta*-metaphysical one—of arguing for or against a way of proceeding, if one were ever to proceed, or arguing that science proceeds in this or that way. Rather, the concrete demand is for a detailed, worked-out account—an account that "goes the distance," in terms of which accounts of particular systems can be formulated, and real-world construction proceed.

For this purpose, with respect to the job of developing an alternative metaphysics, the computational realm has unparalleled advantage. Midway between matter and mind, computation stands in excellent stead as a supply of concrete cases of middling complexity—what in computer science is called an appropriate "vali-

[36] A theory of organization is simply metaphysics with a business plan.

dation suite"—against which to test the mettle of specific metaphysical hypotheses. "Middling" in the sense of neither being so simple as to invite caricature, nor so complex as to defy comprehension. It is the development of a laboratory of this middling sort, half-way between the frictionless pucks and inclined planes of classical mechanics and the full-blooded richness of the human condition, that makes computing such an incredibly important stepping-stone in intellectual history.

Crucially, too, computational examples are examples with which we are as much practically as theoretically familiar (we build systems better than we understand them). Indeed—and by no means insignificantly—there are many famous divides with respect to which computing sits squarely in the middle.

8 Summary

Thus the ante is upped one more time. Not only must an adequate account of computation (any account that meets the three criteria with which we started) include a theory of semantics; it must also include a theory of ontology. Not just intentionality is at stake, in other words; so is metaphysics. But still we are not done. For on top of the foregoing strong conclusions lies an eighth one—if anything even stronger:

C8. Computation is not a subject matter

In spite of everything I said about a comprehensive, empirical, conceptually founded "theory of computing," that is, and in spite of everything I myself have thought for decades, I no longer believe that there is a distinct ontological category of computing or computation, one that will be the subject matter of a deep and explanatory and intellectually satisfying theory. Close and sustained analysis, that is, suggests that the things that Silicon Valley calls computers, the things that perforce *are* computers, do not form a coherent intellectually delimited class. Computers turn out in the end to be rather like cars: objects of inestimable social and political and economic and personal importance, but not in and of themselves, *qua* themselves, the focus of enduring scientific or intellectual inquiry—not, as philosophers would say, a *natural kind*.

Needless to say, this is another extremely strong claim—one over which some readers may be tempted to rise up in arms. At the very least, it is easy to feel massively let down, after all this work. For if I am right, it is not just that we *currently* have no satisfying intellectually productive theory of computing, of the sort I initially set out to find. Nor is it just that, through this whole analysis, I have failed to provide one. It is the even stronger conclusion that such projects will *always* fail; we will *never* have such a theory. So all the previous conclusions must be revised. It is not just that a theory of computation will not *supply* a theory of semantics, for example, as Newell has suggested; or that it will not *replace* a theory of semantics; or even that it will not *depend or rest on* a theory of semantics, as intimated at the end of section 4. It will do none of these things because *there will be no theory of computation at all.*

Given the weight that has been rested on the notion of computation—not just by me, or by computer science, or even by cognitive science, but by the vast majority of the surrounding intellectual landscape—this (like the previous conclusion about ontology) might seem like a negative result. (Among other things, you might conclude I had spent these thirty years in vain.) But in fact there is no cause for grief; for the negativity of the judgment is only superficial, and in fact almost wholly misleading. In fact I believe something almost wholly opposite, which we can label as a (final) conclusion in its own right:

C9. The superficially negative conclusion (that computing is not a subject matter) makes the twentieth-century arrival of computation onto the intellectual scene a *vastly more interesting and important phenomenon than it would otherwise have been.*

On reflection, it emerges that the fact that neither computing nor computation will sustain the development of a theory is by far the most exciting and triumphal conclusion that the computer and cognitive sciences could possibly hope for.

Why so? Because I am not saying that computation-in-the-wild is intrinsically a-theoretical— and thus that there will be no theory of these machines, at all, when day is done. Rather, the claim is that such theory as there is—and I take it that there remains a good chance of such a thing, as much as in any domain of human activity—will not be a theory of *computation* or *computing*. It will not be a theory of computation because *computers per se*, as I have said, do not constitute a distinct, delineated subject matter.

Rather, what computers are, I now believe—and what the considerable and impressive body of practice associated with them amounts to—is neither more nor less than *the full-fledged social construction*[37] *and development of intentional artifacts*. That means that the range of experience and skills and theories and results that have been developed within computer science—astoundingly complex and far-reaching, if still inadequately articulated—is best understood as practical, synthetic, raw material for no less than full theories of causation, semantics, and ontology—that is, for *metaphysics full bore*.

Where does that leave things? Substantively, it leads inexorably to the conclusion that metaphysics, ontology, epistemology, and intentionality are the only integral intellectual subject matters in the vicinity of either computer or cognitive science. Methodologically, it means that our experience with constructing computational (i.e., intentional) systems may open a window onto something to which we would not otherwise have any access: the chance to witness, with our own eyes, how intentional capacities can arise in a "merely" physical mechanism.

It is sobering, in retrospect, to realise that our *preoccupation with the fact that computers are computational* has been the major theoretical block in the way of our understanding how important computers are. They are computational, of course; that much is tautological. But only when we let go of *the conceit that that fact is theoretically important*—only when we let go of the "c-word"—will we finally be able to see, without distraction, and thereby, perhaps, at least partially to understand, how a structured lump of clay can sit up and think.

And so that, for a decade or so, has been my project: to take, from the ashes of computational critique, enough positive morals to serve as the inspiration, basis, and testing ground for an entirely new metaphysics. A story of subjects, a story of objects, a story of reference, a story of history.

For sheer ambition, physics does not hold a candle to computer or cognitive—or rather, as we should now call it, in order to recognise that we are dealing with something on the scale of natural science—*epistemic* or *intentional* science. Hawking (1988) and Weinberg (1994) are wrong. It is we, not the physicists, who must develop a theory of everything.

References

Agre, Philip E. (1996), *Computation and Human Experience*, Cambridge University Press.

Barwise, Jon & Perry, John (1983): *Situations and Attitudes*. Cambridge, Mass.: MIT Press.

Devitt, Michael (1991), "Why Fodor Can't Have It Both Ways," in Loewer, Barry and Rey, Georges (eds.), *Meaning in Mind: Fodor and His Critics*, Oxford: Basil Blackwell, pp. 95–118.

Dretske, Fred I. (1981): *Knowledge and the Flow of Information*, Cambridge, Mass.: MIT Press.

Dreyfus, Herbert (1992), *What Computers Still Can't Do* Cambridge, Mass.: MIT Press.

Fodor, Jerry A. (1975), *The Language of Thought*, Cambridge, Mass.: Harvard University Press.

——— (1980), "Methodological Solipsism Considered as a Research Strategy in Cognitive Psychology," *Behavioral and Brain Sciences*, Vol. 3, No. 1; March 1980, 63–73. Reprinted in Fodor, Jerry: *Representations*, Cambridge, ma.: MIT Press, 1981.

——— (1984), "Semantics, Wisconsin Style," *Synthese*, Vol. 59, pp. 231–250.

——— (1987), *Psychosemantics*, Cambridge: MIT Press.

Halpern, Joe (1987): "Using Reasoning about Knowledge to Analyze Distributed Systems," *Annual Review of Computer Science*, Vol. 2, pp. 37–68.

Harnad, Stevan, (1990) "The Symbol Grounding Problem," *Physica* D Vol. 42, pp. 335–346.

——— (1991), "Other Bodies, Other Minds: A Machine Reincarnation of an Old Philosophical Problem," *Minds and Machines* 1, pp. 43–54.

Haugeland, J. (1978), "The Nature and Plausibility of Cognitivism." *Behavioral and Brain Sciences* 1:215–26.

——— (1982), "Analog and Analog," *Philosophical Topics* (Spring 1981); reprinted in J. I. Biro & Robert W. Shahan, eds., *Mind, Brain, and Function: Essays in the Philosophy of Mind*, Norman, Oklahoma: University of Oklahoma Press (1982), pp. 213–225.

Hawking, Stephen W. (1988) *A Brief History of Time*, Toronto: Bantam Books.

Hutchins, Ed (1995), *Cognition in the Wild*, Cambridge, Mass: MIT Press.

Millikan, Ruth (1984), *Language, Thought, and Other Biological Categories*, Cambridge, Mass.: MIT Press.

——— (1989), "Biosemantics," *Journal of Philosophy* **lxxxvi**:6, June 1989 pp. 281–297.

——— (1990), "Compare and Contrast Dretske, Fodor, and Millikan on Teleosemantics," *Philosophical Topics* 18:2, Fall 1990, pp. 151–161.

[37] Social construction not as the label for a metaphysical stance, but in the literal sense that we build them.

Newell, Alan and Simon, Herbert A. (1976): "Computer Science as Empirical Inquiry: Symbols and Search," *Communications of the Association for Computing Machinery*, vol. 19 (March 1976), No. 3, pp. 113–126. Reprinted in John Haugeland, ed., *Mind Design*, Cambridge, Mass: MIT Press (1981), pp. 35–66.

Searle, John (1980): "Minds, Brains, and Programs," *Behavioral and Brain Sciences*, Vol. 3, No. 3, Sept. 1980, pp. 417–458.

Shannon, Claude E. & Weaver, Warren (1949), *The Mathematical Theory of Communication*, Urbana, Illinois: The University of Illinois Press.

Smith, Brian Cantwell (1996), *On the Origin of Objects*, Cambridge, Mass: MIT Press.

——— (in press), Reply to Dan Dennett. Hugh Clapin (ed.), *Philosophers of Mental Representation*, Oxford University Press, forthcoming 2002.

——— (forthcoming), *The Age of Significance: Volumes* i–vii.

van Gelder, T. (1996), Dynamics and Cognition, in John Haugeland (ed. 1996), 421–450. Loosely based on "What Might Cognition Be, if not Computation?" *Journal of Philosophy*, 91, 345–381

Weinberg, S. (1994), *Dreams of a Final Theory*, Vintage Books.

— Were this page been blank, that would have been unintentional —

B · Reflection

— Were this page been blank, that would have been unintentional —

Procedural Reflection in Programming Languages[†]

2a — Abstracts, Preface, and Prologue

1 Abstract

We show how a computational system can be constructed to "reason," effectively and consequentially, about its own inferential processes.[a] The analysis proceeds in two parts. First, we consider the general question of computational semantics, rejecting traditional approaches, and arguing that the *declarative* and *procedural* aspects of computational symbols (what they stand for, and what behaviour they engender) should be analysed *independently*, in order that they may be coherently related. Second, we investigate *self-referential* behaviour in computational processes, and show how to embed an effective procedural model of a computational calculus within that calculus (a model not unlike a meta-circular interpreter, but connected to the fundamental operations of the machine in such a way as to provide, at any point in a computation, fully articulated descriptions of the state of that computation, for inspection and possible modification). In terms of the theories that result from these investigations, we present a general architecture for **procedurally reflective** processes, able to shift smoothly between dealing with a given subject domain, and dealing with their own reasoning processes over that domain.

An instance of the general solution is worked out in the context of an applicative language. Specifically, we present three successive

[†] "Prologue", Brian Cantwell Smith, *Procedural Reflection in Programming Languages*, doctoral dissertation submitted Jan 25, 1982 in the Laboratory for Computer Science, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (MIT), Cambridge, MA. Reprinted as Technical Report MIT-LCS-TR-272, Laboratory for Computer Science, MIT, Cambridge, MA, 1982. Available at:
http://publications.csail.mit.edu/lcs/specpub.php?id=840

[a] Note: footnotes to the original versions of these Abstracts, the Preface, and the Prologue are numbered sequentially (1–14); footnotes added for this publication are identified by letter (a–m).

dialects of Lisp: **1-Lisp**, a distillation of current practice, for comparison purposes; **2-Lisp**, a dialect constructed in terms of our rationalised semantics, in which the concept of evaluation is rejected in favour of independent notions of *simplification* and *reference*, and in which the respective categories of notation, structure, semantics, and behaviour arc strictly aligned; and **3-Lisp**, an extension of 2-Lisp endowed with reflective powers.

2 Extended Abstract

We show how a computational system can be constructed to "reason" effectively and consequentially about its own inference processes. Our approach is to analyse *self-referential* behaviour in computational systems, and to propose a theory of **procedural reflection** that enables any programming language to be extended in such a way as to support programs able to access and manipulate structural descriptions of their own operations and structures. In particular, one must encode an explicit theory of such a system within the structures of the system, and then connect that theory to the fundamental operations of the system in such a way as to support three primitive behaviours. First, at any point in the course of a computation, fully articulated descriptions of the state of the reasoning process must be available for inspection and modification. Second, it must be possible at any point to resume an arbitrary computation in accord with such (possibly modified) theory-relative descriptions. Third, procedures that reason with descriptions of the processor state must themselves be subject to description and review, to arbitrary depth. Such **reflective** abilities allow a process to shift smoothly between dealing with a given subject domain, and dealing with its own reasoning processes over that domain.

Crucial in the development of this theory is a comparison of the respective semantics of programming languages (such as Lisp and Algol) and declarative languages (such as logic and the l-calculus); we argue that unifying these traditionally separate disciplines clarifies both, and suggests a simple and natural approach to the question of procedural reflection. More specifically, the semantical analysis of computational systems should comprise independent formulations of **declarative import** (what symbols stand for) and **procedural consequence** (what effects and results are engendered by processing them), although the two semantical treatments may, because of side-effect interactions, have to be formulated in conjunction. When this approach is applied to a functional language it is shown that the traditional notion of *evaluation* is confusing and confused, and must be rejected in favour of independent notions of *reference* and *simplification*. In addition, we defend a standard of **category alignment**: there should be a systematic correspondence between the respective categories of notation, abstract structure, declarative semantics, and procedural consequence (a mandate satisfied by no extant procedural formalism). It is shown how a clarification of these prior semantical and aesthetic issues enables a **procedurally reflective** dialect to be clearly defined and readily constructed.

An instance of the general solution is worked out in the context of an applicative language, where the question reduces to one of defining an interpreted calculus able to inspect and affect its own interpretation. In particular, we consider three successive dialects of Lisp: **1-Lisp**, a distillation of current practice for comparison purposes; **2-Lisp**, a dialect *categorically* and *semantically rationalised* with respect to an explicit theory of declarative semantics for s-expressions; and **3-Lisp**, a derivative of 2-Lisp endowed with full reflective powers. 1-Lisp, like all Lisp dialects in current use, is at heart a *first-order* language, employing meta-syntactic facilities and dynamic variable scoping protocols to partially mimic higher-order functionality. 2-Lisp like Scheme and the l-calculus, is higher-order: it supports arbitrary function designators in argument position, is lexically scoped, and treats the function position of an application in a standard extensional manner. Unlike Scheme, however, the 2-Lisp processor is based on a regimen of *normalisation*, taking each expression into a normal-form co-designator of its referent, where the notion of *normal-form* is in part defined with respect to that referent's semantic type, not (as in the case of the l-calculus) solely in terms of the further non-applicability of a set of syntactic reduction rules. 2-Lisp normal-form designators are environment-independent and side-effect free; thus the concept of a *closure* can be reconstructed as a *normal-form function designator*. In addition, since normalisation is a form of simplification, and is therefore *designation-preserving*, meta-structural expressions are not de-referenced upon normalisation, as they are when evaluated. Thus we say that the 2-Lisp processor is *semantically flat,* since it stays at a semantically fixed level (although explicit referencing and de-referencing primitives are also provided, to facilitate explicit level shifts). Finally, because of its category alignment, *argument objectification* (the ability to apply functions to a sequence of arguments designated collectively by a single term) can be treated in the 2-Lisp base-level language, without requiring resort to meta-structural machinery.

3-Lisp is straightforwardly defined as an extension of 2-Lisp, with respect to an explicitly articulated procedural theory of 3-Lisp embedded in 3-Lisp structures. This embedded theory, called the **reflective model**, though superficially resembling a meta-circular interpreter, is causally connected to the workings of the underlying calculus in crucial and primitive ways. Specifically, *reflective procedures* are supported that bind as arguments (designators of) the continuation and environment structure of the processor that *would have been in effect at the moment the reflective procedure was called, had the machine been running all along in virtue of the explicit processing of that reflective model.*[a] Because reflection may recurse arbitrarily, 3-Lisp is most simply defined as an infinite tower of 3-Lisp processes, each engendering the process immediately below it. Under such an account, the use of reflective procedures amounts to running programs at arbitrary levels in this reflective hierarchy. Both a straightforward implementation and a conceptual analysis are provided to demonstrate that such a machine is nevertheless finite.

The 3-Lisp reflective model unifies three programming language concepts that have formerly been viewed as independent: meta-circular interpreters, explicit names for the primitive interpretive procedures (`EVAL` and `APPLY` in standard Lisp dialects), and procedures that access the state of the implementation (typically provided, as part of a programming environment, for debugging purposes). We show how all such behaviours can be defined within a pure version of 3-Lisp (i.e., independent of implementation), since all aspects of the state of any 3-Lisp process are available, with sufficient reflection, as objectified entities within the 3-Lisp **structural field**.

[a] Emphasis added.

3 Preface

The possibility of constructing a reflective calculus first struck me in June 1976, at the Xerox Palo Alto Research Center (PARC), where I was spending a summer working with the KRL representation language of Bobrow and Winograd.[1] As an exercise to learn the new language, I had embarked on the project cf representing KRL in KRL; it seemed to me that this "double-barreled" approach, in which I would have both to *use* and to *mention* the language, would be a particularly efficient way to unravel its intricacies. Though that exercise was ultimately abandoned, I stayed with it long enough to become intrigued by the thought that one might build a system that was self-descriptive in an important way (certainly in a way in which my KRL project was *not*). More specifically, I could dimly envisage a computational system in which what happened took effect in virtue of declarative descriptions of what was to happen, and in which the internal structural conditions were represented in declarative descriptions of those internal structural conditions. In such a system a program could with equal ease access all the basic operations and structures either directly or in terms of completely (and automatically) articulated descriptions of them. The idea seemed to me rather simple (as it still does); furthermore, for a variety of reasons I thought that such a reflective calculus could *itself* be rather simple—in some important ways simpler than a non-reflective formalism (this too I still believe). *Designing* such a formalism, however, no longer seems as straightforward as I thought at the time; this dissertation should be viewed as the first report emerging from the research project that ensued.

Most of the five years since 1976 have been devoted to initial versions of my specification of such a language, called **Mantiq**, based on these original hunches.[b] As mentioned in the first paragraph of chapter 1, there are various non-trivial goals that must be met by the designer of any such formalism, including at least a tentative solution to the knowledge representation problem. Furthermore, in the course of its development, MANTIQ has come to

[1] 'KRL' for 'Knowledge Representation Language; see Bobrow and Winograd (1977) and Bobrow et al. (1977).

[b] «Say something about the provenance of the name, and the fate of the project.»

rest on some additional hypotheses above and beyond those mentioned above (including, for example, a sense that it will be possible within a computational setting to construct a formalism in which syntactic identity and intensional identity can be identified, given some appropriate, but *independently* specified, theory of intensionality[c]). Probably the major portion of my attention to date has focused on these intensional aspects of the MANTIQ architecture.

It was clear from the outset that no dialect of Lisp (or of any other purely procedural calculus) could serve as a full reflective formalism; purely declarative languages like logic or the l-calculus were dismissed for similar reasons.[d] In February of 1981, however, I decided that it would be worth focusing on Lisp, by way of an example, in order to work out the details of a specific subset of the issues with which MANTIQ would have to contend. In particular, I recognised that many of the questions of reflection could be profitably studied in a (limited) procedural dialect, in ways that would ultimately illuminate the larger programme. Furthermore, to the extent that Lisp could serve as a theoretical vehicle, it seemed a good project; it would be much easier to develop, and even more so to communicate, solutions in a formalism at least partially understood.

The time from the original decision to look at procedural reflection (and its concomitant emphasis on semantics—I realised from investigations of MANTIQ that semantics would come to the fore in all aspects of the overall enterprise), to a working implementation of 3-Lisp, was only a few weeks. Articulating why 3-Lisp was the way it was, however—i.e., formulating in plain English the concepts and categories on which the design was founded—required quite intensive work for the remainder of the year. A first draft of the dissertation was completed at the end of December 1981; the implementation remained essentially unchanged during the course of this writing (the only substantive alteration was the idea

[c] «Give an example of what this meant and means, why it is important, what it would require ⟦substantial relaxation algorithms⟧, why it has not yet been achieved, why it is still something worth pursuing, etc. May need to explain what 'intensionality' means.»

[d] «Refer to—and perhaps include, if I still have it?—the reflective l-calculus that I subsequently defined ⟦for Barwise⟧»

of treating recursion in terms of explicit Y-operators). Thus—and I suspect there is nothing unusual in this experience—formulating an idea required approximately ten times more work than embodying it in a machine; perhaps more surprisingly, all of that effort in formulation occurred *after* the implementation was complete, [and led to no revisions in the basic design]. We sometimes hear that writing computer programs is intellectually hygienic because it requires that we make our ideas completely explicit. I have come to disagree rather fundamentally with this view. Certainly writing a program does not force one to one make one's ideas *articulate,* although it is a useful first step. More seriously, however, it is often the case that the organising principles and fundamental insights contributing to the coherence of a program are not explicitly encoded within the structures comprising that program. The theory of declarative semantics embodied in 3-Lisp, for example, was initially tacit—a fact perhaps to be expected, since only procedural consequence is explicitly encoded in an implementation. Curiously, this is one of the reasons that building a fully reflective formalism (as opposed to the limited procedurally reflective languages considered here) is difficult: in order to build a general reflective calculus, one must embed within it a fully articulated theory of one's understanding of it. This will take some time.

4 Prologue

It is a striking fact about human cognition that we can think not only about the world around us, but also about our ideas, our actions, our feelings, our past experience. This ability to **reflect** lies behind much of the subtlety and flexibility with which we deal with the world; it is an essential part of mastering new skills, of reacting to unexpected circumstances, of short-range and long-range planning, of recovering from mistakes, of extrapolating from past experience, and so on and so forth. Reflective thinking characterises mundane practical matters and delicate theoretical distinctions. We have all paused to review past circumstances, such as conversations with guests or strangers, to consider the appropriateness of our behaviour. We can remember times when we stopped and consciously decided to consider a set of options, say when confronted with a fire or other emergency. We understand when someone tells us to believe everything a friend tells us, unless we know otherwise. In the course of philosophical discussion we can agree to distinguish views we believe to be true from those we have no reason to believe are false. In all these cases the subject matter of our contemplation at the moment of reflection includes our remembered experience, our private thoughts, and our reasoning patterns.

The power and universality of reflective thinking has caught the attention of the cognitive science community—indeed, once alerted to this aspect of human behaviour, theorists find evidence of it almost everywhere. Though no one can yet say just what it comes to, crucial ingredients would seem to be the ability to recall memories of a world experienced in the past and of one's own participation in that world, the ability to think about a phenomenal world, hypothetical or actual, that is not currently being experienced (an ability presumably mediated by our knowledge and belief), and a certain kind of true self-reference: the ability to consider both one's actions and the workings of one's own mind. This last aspect—the self-referential aspect of reflective thought—has sparked particular interest for cognitive theorists, both in psychology (under the label *meta-cognition)* and in artificial intelligence (in the design of computational systems possessing inchoate reflective powers, particularly as evidenced in a collection of ideas loosely allied in their use of the term "meta": meta-level rules, meta-descriptions, and so forth).

In artificial intelligence, the focus on computational forms of self-referential reflective reasoning has become particnlarly central. Although the task of endowing computational systems with subtlety and flexibility has proved difficult, we have had some success in developing systems with a moderate grasp of certain domains: electronics, bacteremia, simple mechanical systems, etc. One of the most recalcitrant problems, however, has been that of developing flexibility and modularity (in some cases even simple effectiveness) in the reasoning processes that use this world knowledge. Though it has been possible to construct programs that perform a specific kind of reasoning task (say, checking a circuit or parsing a subset of natural language syntax), there has been less success in simulating "common sense," or in developing programs able to figure out what to do, and how to do it, in either general or novel situations. If the course of reasoning—if the problem solving strategies and the hypothesis formation behaviour—could *itself* be treated as a valid subject domain in its own right, then (at least so the idea goes) it might be possible to construct systems that manifested the same modularity about their own thought processes that they manifest about their primary subject domains. A simple example might be an electronics "expert" able to choose an appropriate method of tackling a particular circuit, depending on a variety of questions about the relationship between its own capacities and the problem at hand: whether the task was primarily one of design or analysis or repair, what strategies and skills it knew it had in such areas, how confident it was in the relevance of specific approaches based on, say, the complexity of the circuit, or on how similar it looked compared with circuits it already knew. Expert human problem-solvers clearly demonstrate such reflective abilities, and it appears more and more certain that powerful computational problem solvers will have to possess them as well.

No one would expect potent skills to arise automatically in a reflective system; the mere *ability* to reason about the reasoning process will not magically yield systems able to reflect in powerful and flexible ways. On the other hand, the demonstration of such an ability is clearly a pre-requisite to its effective utilisation. Furthermore, many reasons are advanced in support of reflection, as well as the primary one (the hope of building a system able to decide how to structure the pattern of its own reasoning). It has been argued, for example, that it would be easier to construct powerful systems in the first place (it would seem you could almost *tell them* how to think), to interact with them when they fail, to trust them if they could report on how they arrive at their decisions, to give them "advice" about how to improve or discriminate, as well as to provide them with their own strategies for reacting to their history and experience.

There is even, as part of the general excitement, a tentative suggestion on how such a self-referential reflective process might be constructed. This suggestion—nowhere argued but clearly in evidence in several recent proposals—is a particular instance of a general hypothesis, adopted by most A.I. researchers, that we will call the **Knowledge Representation Hypothesis**. It is widely held in computational circles that any process capable of reasoning intelligently about the world must consist in part of a field of structures, of a roughly linguistic sort, which in some fashion *represent* whatever knowledge and beliefs the process may be said to possess. For example, according to this view, since I know that the sun sets each evening, my "mind" must contain (among other things) a language-like or symbolic structure that represents this fact, inscribed in some kind of internal code. There are various assumptions that go along with this view: there is for one thing presumed to be an internal process that "runs over" or "computes with" these representational structures, in such a way that the intelligent behaviour of the whole results from the interaction of parts. In addition, this ingredient process is required to react only to the "form" or "shape" of these mental representations, without regard to what they mean or represent—this is the substance of the claim that computation involves *formal* symbol manipulation. Thus my thought that, for example, the sun will soon set, would be taken to emerge from an interaction in my mind between an ingredient process and the shape or "spelling" of various internal structures representing my knowledge that the sun does regularly set each evening, that it is currently tea time, and so forth.

The knowledge representation hypothesis may be summarised as follows:

> **Knowledge Representation Hypothesis:** *Any mechanically embodied intelligent process will be comprised of structural ingredients that (a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and (b) independent of such external semantical attribution, play a formal but causal and essential role in engendering the behaviour that manifests that knowledge.*

Thus for example if we felt disposed to say that some process knew that dinosaurs were warm-blooded, then we would find (according, presumably, to the best explanation of how that process worked) that a certain computational ingredient in that process was understood as *representing* the (propositional) fact that dinosaurs were warm-blooded, and furthermore, that this very ingredient played a role, independent of our understanding of it as representational, in leading the process to behave in whatever way inspired us to say that it knew that fact. Presumably we would be convinced by the manner in which the process answered certain questions about their likely habitat, by assumptions it made about other aspects of their existence, by postures it adopted on suggestions as to why they may have become extinct, etc.

A careful analysis will show that. to the extent that we can make sense of it, this view that *knowing is representational* is far less evident—and perhaps, therefore, far more interesting—than is commonly believed. To do it justice requires considerable care: accounts in cognitive psychology and the philosophy of mind tend to founder on simplistic models of computation. and artificial intelligence treatments often lack the theoretical rigour necessary to bring the essence of the idea into plain view. Nonetheless, conclusion or hypothesis, it permeates current theories of mind, and has in particular led researchers in artificial intelligence to propose a spate of computational languages and calculi designed to underwrite such representation. The common goal is of course not so much to speculate on what is actually represented in any particular situation as to uncover the general and categorical form of such representation. Thus no one would suggest how anyone actually represents facts about tea and sunsets: rather, they might posit the general form in which such beliefs would be "written" (along with other beliefs, such as that Lhasa is in Tibet, and that p is an irrational number). Constraining all plausible suggestions, however, is the requirement that they must be able to demonstrate how a particular thought could emerge from such representations—this is a crucial meta-theoretic characteristic of artificial intelligence research. It is traditionally considered insufficient merely to propose true theories that do not enable some causally effective mechanical embodiment. The standard against which such theories must ultimately judged, in other words, is whether they will serve to underwrite the construction of demonstrable, behaving artefacts. Under this general rubric knowledge representation efforts differ markedly in scope, in approach, and in detail; they differ on such crucial questions as whether or not the mental structure are modality specific (one for visual memory, another for verbal, for example). In spite of such differences, however, they manifest the shared hope that an attainable first step towards a full theory of mind will be the discovery of something like the structure of the "mechanical mentalese" in which our beliefs are inscribed.

It is natural to ask whether the knowledge representation hypothesis deserves our endorsement, but this is not the place to pursue that difficult question. Before it can fairly be asked, we would have to distinguish a strong version claiming that knowing is *necessarily* representational from a weaker version claiming merely that it is *possible* to build a representational knower. We would run straight into all the much-discussed but virtually intractable questions about what would be required to convince us that an artificially constructed process exhibited intelligent behaviour. We would certainly need a definition of the word 'represent,' about which we will subsequently have a good deal to say. Given the current (minimal) state of our understanding, I myself see no reason to subscribe to the strong view, and remain skeptical of the weak version as well.[b] But one of the most difficult questions is merely to ascertain what the hypothesis is actually saying—thus my interest in representation is more a concern to *make it clear* than it is to defend or deny it The entire present investigation, therefore, will be pursued under this hypothesis, not because we grant it our al-

[b] … «talk about this» …

legiance, but merely because it deserves our attention.

Given the representation hypothesis, the suggestion as to how to build self-reflective systems—a suggestion we will call the *Reflection Hypothesis*—can be summarised as follows:[h]

> **Reflection Hypothesis:** *In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures.*

Thus the task of building a computationally reflective system is thought to reduce to, or at any rate to include, the task of providing a system with formal representations of its own constitution and behaviour. Hence a system able to imagine a world where unicorns have wings would have to construct formal representations of that fact; a system considering the adoption of a hypothesis-and-test style of investigation would have to construct formal structures representing such an inference regime.

Whatever its merit, there is ample evidence that researchers arc taken with this view. Systems such as Weyhrauch's FOL, Doyle's TMS, McCarthy's ADVICE-TAKER, Hayes' GOLUM, and Davis' TERESIUS arc particularly explicit exemplars of just such an approach.[2] In Weyhrauch's system, for example, sentences in first-order logic arc constructed that axiomatize the behaviour of the Lisp procedures used in the course of the computation (FOL is a prime example of the dual-calculus approach mentioned earlier). In Doyle's systems, explicit representations of the dependencies between beliefs and of the "reasons" the system accepts a conclusion play a causal role in the inferential process. Similar remarks hold for the other projects mentioned, as well as for a variety of other current research. In addition, it turns out on scrutiny that a great deal of current computational practice can be seen as dealing, in one way or another, with reflective abilities, particularly as exem-

[h] «Note that the numbered indentation of the following paragraphs has been added, for clarity.»

[2] Weyhrauch (1978), Doyle (1979), McCarthy (1968), Hayes (1979), and Davis (1980a), respectively.

plified by computational structures representing other computational structures. We constantly encounter examples: the widespread use of macros in Lisp, the use of meta-level structures in representation languages, the use of explicit non-monotonic inference rules, the popularity of meta-level rules in planning systems.[3] Such a list can be extended indefinitely; in a recent symposium Brachman reported that the love affair with *"meta-level reasoning"* was the most important theme of knowledge representation research in the last decade.[4]

4a The Relationship Between Reflection & Representation

The manner in which this discussion has been presented so far would seem to imply that the interest in *reflection* and the adoption of a *representational* stance are theoretically independent positions. I have argued in this way for a reason: to make clear that the two subjects are not the same. There is no *a priori* reason to believe that even a fully representational system should in any way be reflective or able to make anything approximating a reference to itself; similarly, there is no *proof* that a powerfully self-referential system need be constructed of representations. However—and this is the crux of the matter—the reason to raise both issues together is that they are surely, in some sense, related. If nothing else, the word 'representation' comes from 're' plus 'present', and the ability to *re-present* a world to itself is undeniably a crucial, if not *the* crucial, ingredient in reflective thought. If I reflect on my childhood, I re-present to myself my school and the rooms of my house; if I reflect on what I will do tomorrow, I bring into the view of my mind's eye the self I imagine that tomorrow I will be. If we take "representation" to describe an *ability* rather than a *structure,* reflection surely involves representation (although—and this should be kept clearly in mind—the "representation" of the knowledge representation hypothesis refers to ingredient structures, not to an activity).

It is helpful to look at the historical association between these

[3] For a discussion of macros see the various sources on Lisp mentioned in note 16 of chapter 1; meta-level rules in representation were discussed in Brachman and Smith (1980); for a collection of papers on non-monotonic reasoning see Bobrow (1980); macros are discussed in Pitman (1980).

[4] Brachman (1980).

ideas, as well to search for commonalities in content. In the early days of artificial intelligence, a search for the general patterns of intelligent reasoning led to the development of such general systems as Newell and Simon's GPS, predicate logic theorem provers, and so forth.[5] The descriptions of the subject domains were minimal but were nonetheless primarily declarative, particularly in the case of the systems based on logic. However it proved difficult to make such general systems effective in particular cases: so much of the "expertise" involved in problem solving seems domain and task specific. In reaction against such generality, therefore, a *procedural* approach emerged in which the primary focus was on the manipulation and reasoning about specific problems in simple worlds.[6] Though the procedural approach in many ways solved the problem of undirected inferential meandering, it too had problems: it proved difficult to endow systems with much generality or modularity when they were simply constituted of procedures designed to manifest certain particular skills. In reaction to such brittle and parochial behaviour, researchers turned instead to the development of processes designed to work over general representations of the objects and categories of the world in which the process was designed to be embedded. Thus the *representation hypothesis* emerged in the attempt to endow systems with generality, modularity, flexibility, and so forth with respect to the embedding world, but to retain a procedural effectiveness in the control component.[7] In other words, in terms of our main discussion, representation as a method emerged as a solution to the problem of providing general and flexible ways of reflecting (not self-referentially) about the world.

Systems based on the representational approach—and it is fair to say that most of the current "expert systems" are in this tradition—have been relatively successful in certain respects, but a major lingering problem has been a narrowness and inflexibility regarding the style of reasoning these systems employ in using these

[5] Newell and Simon (1963); Newell and Simon (1956).

[6] The proceduralist view was represented particularly by a spate of dissertations emerging from MIT at the beginning of the 1970s; see for example Winograd (1972), Hewitt (1972), Sussman et al. (1971), etc.

[7] See Minsky (1975), Winograd (1975), and all of the systems reported in Brachman and Smith (1980).

representational structures. This inflexibility in *reasoning* is strikingly parallel to the inflexibility in *knowledge* that led to the first round of representational systems; researchers have therefore suggested that we need reflective systems able to deal with their own constitutions as well as with the worlds they inhabit. In other words, since the *style* of the problem is so parallel to that just sketched, it has seemed that another application of the same medicine might be appropriate. If we could inscribe general knowledge about how to reason in a variety of circumstances in the "mentalese" of these systems, it might be possible to design a relatively simpler inferential regime over this "meta-knowledge about reasoning," thereby engendering a flexibility and modularity regarding reasoning, just as the first representational work engendered a flexibility and modularity about the process's embedding world.

There are problems, however, in too quick an association between the two ideas, not the least of which is the question of to *whom* these various forms of re-presentation are being directed. In the normal case—that is to say, in the typical computational process built under the aegis of the knowledge representation hypothesis—a process is constituted from symbols that we as external theorists take to be representational structures; they are visible *only to the ingredient interpretive process [that is just part] of the whole,* and they are visible to that constituent process *only formally* (this is the basic claim of computation). Thus the interpreter can see them, though it is blind to the fact of their being representations. (In fact it is almost a great joke that the blindly formal ingredient process should be called an *interpreter*: when the Lisp interpreter evaluates the expression '`(+ 2 3)`' and returns the result '`6`', the last thing it knows is that the numeral '`2`' denotes the number *two*.[c])

Whatever is the case with the ingredient process, there is no reason to suppose that the representational structures are visible to the whole constituted process *at all,* formally or informally. That process is made out of them; there is no more *a priori* reason to suppose that they are accessible to its inspection than to suppose that a camera could take a picture of its own shutter—no more reason to suppose it is even a coherent possibility than to say that France is near Marseilles. Current practice should overwhelm-

[c] «Talk about the 100 Billion Lines coming later … »

ingly convince us of this point: what is as tacit—what is as thoroughly lacking in self-knowledge—as the typical modern computer system?[d]

The point of the argument here is not to prove that one *cannot* make such structures accessible—that one *cannot* make a representational reflective system—but to make clear that two ideas are involved. Furthermore, they are different in kind: one (representation) is a possibly powerful *method* for the construction of systems; the other (reflection) is a kind of *behaviour* we are asking our systems to exhibit. It remains a question whether the representational method will prove useful in the pursuit of the goal of reflective behaviour.

[Answering that question], in a nutshell, is our overall project.

4b The Theoretical Backdrop

It takes only a moment's consideration of such questions as the relationship between representation and reflection to recognise that the current state of our understanding of such subjects is terribly inadequate. In spite of the general excitement about reflection, self-reference, and computational representation, no one has presented an underlying theory of any of these issues. The reason is simple: we are so lacking in adequate theories of the surrounding territory that, without considerable preliminary work, cogent definitions cannot even be attempted. Consider for example the case regarding self-referential reflection, where just a few examples will make this clear.

1. From the fact that a reflective system *A* is implemented in system *B*, it docs not follow that system *B* is thereby rendered reflective (for example, in this dissertation I will present a partially-reflective dialect of Lisp that I have implemented on a Digital Systems Corporation PDP-10,[e] but the PDP-10 is not itself reflective). Hence even a *definition* of reflection will have to be backed by theoretica1 apparatus capable of distinguishing between one abstract machine and another in which the first is implemented—something we

[d] «Talk about response to Charles Taylor…»

[e] «Explain character and historical role»

are not yet able to do.[f]

2. The notion seems to require of a computational process, and (if we subscribe to the representational hypothesis) of its interpreter, that in reflecting it "back off" one level of reference, and we lack theories both of interpreters in general, and of computational reference in particular.[g]

3. Theories of computational interpretation will be required to clarify the confusion mentioned above regarding the relationship between reflection and representation: for a system to reflect it must re-present *for itself* its mental states; it is not sufficient for it to comprise a set of formal representations inspected *by its interpreter*. This is a distinction we encounter again and again; a failure to make it is the most common error in discussions of the plausibility of artificial intelligence from those outside the computational community, derailing the arguments of such thinkers as Searle and Fodor.[8]

4. Theories of reference will be required in order to make sense of the question of what a computational process is "thinking" about at all, whether reflective or not (for example. it may be easy to claim that when a program is manipulating data structures representing women's votes that the process as a whole is "thinking about suffrage," but what is the process thinking about when the interpreter is expanding a macro definition?).

5. Finally, if the search for reflection is taken up too enthusiastically, one is in danger of interpreting everything as evidence of reflective thinking, since what may not be reflective *explicitly* can usually be treated as *implicitly* reflective (especially given a little imagination on the part of the theorist). However we lack general guidelines on how to distinguish explicit from implicit aspects of computational structures.

[f] «Talk about subsequent theoretical work this points towards ...»

[g] «Talk about 2-Lisp semantics, and how uninterpretable that attempt was ...»

[8] Searle (1980), Fodor (1978 and 1980). «Also point forwards to 100 Billion»

Nor is our grasp of the representational question any clearer; a serious difficulty, especially since the representational endeavour has received much more attention than has reflection. Evidence of this lack can be seen in the fact that, in spite of an approximate consensus regarding the general form of the task, and substantial effort on its behalf, no representation scheme yet proposed has won substantial acceptance in the field. Again this is due at least in part to the simple absence of adequate theoretical foundations in terms of which to formulate either enterprise or solution. We do not have theories of either representation or computation in terms of which to define the terms of art currently employed in their pursuit (*representation, implementation, interpretation, control structure. data structure, inheritance,* and so forth), and are consequently without any well-specified account of what it would be to succeed, let alone of what to investigate, or of how to proceed.[i] Numerous related theories have been developed (model theories for logic, theories of semantics for programming languages, and so forth), but they do not address the issues of knowledge representation directly, and it is surprisingly difficult to weave their various insights into a single coherent whole.

The representational consensus alluded to above, in other words, is widespread but vague; disagreements emerge on every conceivable technical point, as was demonstrated in a recent survey of the field.[9] To begin with, the central notion of "representation" remains notoriously unspecified: in spite of the intuitions mentioned above, there is remarkably little agreement on whether a representation must "re-present" in any constrained way (like an image or copy), or whether the word is synonymous with such general terms as "sign" or "symbol". A further confusion is shown by an inconsistency in usage as to what representation is a relationship between. The sub-discipline is known as the *representation of knowledge,* but in the survey just mentioned by far the majority of the respondents (to the surprise of this author) claimed to use the word, albeit in a wide variety of ways, as between formal symbols *and the world about which the process is designed to reason.* Thus a KLONE structure might be said to *represent Don Quixote*

[i] Again, point forward to AOS.

[9] Brachman and Smith (1980).

tilting at a windmill; it would not be taken as representing *the fact or proposition of this activity*. In other words the majority opinion is not that we are *representing knowledge* at all, but rather, as we put it above, that *knowing is representational*.[10]

In addition, we have only a dim understanding of the relationship that holds between the purported representational structures and the ingredient process that interprets them. This relates to the crucial distinction between that interpreting process and the whole process of which it is an ingredient (whereas it is *I* who thinks of sunsets, it is at best a *constituent of my mind* that inspects a mental representation[j]). Furthermore, there are terminological confusions: the word 'semantics' is applied to a variety of concerns, ranging from how natural language is translated into the representational structures, to what those structures represent, to how they impinge on the rational policies of the "mind" of which they are a part, to what functions are computed by the interpreting process, etc.[k] The term 'interpretation' (to take another example) has two relatively well-specified but quite independent meanings, one of computational origin, the other more philosophical; how the two relate remains so far unexplicated, although, as was just mentioned, they are strikingly distinct.

Unfortunately, such general terminological problems are just the tip of an iceberg. When we consider our specific representational proposals, we are faced with a plethora of apparently incomparable technical words and phrases. *Node, frame, unit, concept, schema, script, pattern, class,* and *plan,* for example, are all popular terms with similar connotations and ill-defined meaning.[11] The theoretical situation (this may not be so harmful in terms of more practical goals) is further hindered by the tendency for representational research to be reported in a rather demonstrative fashion: researchers typically exhibit particular formal systems that (often quite impressively) embody their insights, but that

[10] See the introduction to Brachman and Smith (1980).

[j] «Point forward to internal-representation-registrational item (whatever I end up calling it)»

[k] «Point forward to 100 Billion»

[11] References on *node, frame, unit, concept, schema, script, pattern, class,* and *plan* can be found in the various references provided in Brachman and Smith (1980).

are defined using formal terms peculiar to the system at hand. We are left on our own to induce the relevant generalities and to locate them in our evolving conception of the representation enterprise as a whole. Furthermore, such practice makes comparison and discussion of technical details always problematic and often impossible, defeating attempts to build on previous work.

This lack of grounding and focus has not passed unnoticed: in various quarters one hears the suggestion that, unless severely constrained, the entire representation enterprise may be ill-conceived—that we should turn instead to considerations of particular epistemological issues (such as how we reason about, say, liquids or actions), and should use as our technical base the traditional formal systems (logic, Lisp, and so forth) that representation schemes were originally designed to replace.[12] In defense of this view two kinds of argument are often advanced. The first is that questions about the *central* cognitive faculty are at the very least premature, and more seriously may for principled reasons never succumb to the kind of rigorous scientific analysis that characterizes recent studies of the *peripheral* aspects of mind: vision, audition, grammar, manipulation, and so forth.[13] The other argument is that logic as developed by the logicians is in itself sufficient; that all we need is a set of ideas about what axioms and inference protocols are best to adopt.[14] But such doubts cannot be said to have deterred the whole of the community: the survey just mentioned lists more than thirty new representation systems under active development.

The strength of this persistence is worth noting, especially in connection with the theoretical difficulties just sketched. There can be no doubt that there are scores of difficult problems: we have just barely touched on some of the most striking. But it would be a mistake to conclude in discouragement that the *enterprise* is doomed, or to retreat to the meta-theoretic stability of adjacent fields (like proof theory, model theory, programming language

[12] See in particular Hayes (1978).

[13] The distinction between central and peripheral aspects of mind is articulated in Nilsson (1981); on the impossibility of central AI (Nilsson himself feels that the central faculty will quite definitely succumb to AI's techniques) see Dreyfus (1972) and Fodor (1980 and forthcoming).

[14] Nilsson (1981).

semantics, and so forth). The moral is at once more difficult and yet more hopeful. What is demanded is that we stay true to these undeniably powerful ideas, and attempt to develop adequate theoretical structures on this home ground. It is true that any satisfactory theory of computational reflection must ultimately rest, more or less explicitly, on theories of computation, of intensionality, of objectification, of semantics and reference, of implicitness, of formality, of computation, of interpretation, of representation, and so forth. On the other hand as a community we have a great deal of practice that often embodies intuitions that we are unable to formulate coherently. The wealth of programs and systems we have built often betray—sometimes in surprising ways—patterns and insights that eluded our conscious thoughts in the course of their development. What is mandated is a *rational reconstruction* of those intuitions and of that practice.

In the case of designing reflective systems, such a reconstruction is curiously urgent. In fact this long introductory story ends with an odd twist—one that "ups the ante" in the search for a carefully formulated theory, and suggests that practical progress will be impeded until we take up the theoretical task. In general, it is of course possible (some would even advocate this approach) to build an instance of a class of artefact before formulating a theory of it. The era of sail boats, it has often been pointed out, was already drawing to a close just as the theory of airfoils and lift was being formulated—the [very] theory that, at least at the present time, best explains how those sailboats worked. However there are a number of reasons why such an approach may be ruled out in the present case. For one thing, in constructing a reflective calculus one must support arbitrary levels of meta-knowledge and self-modelling, and it is self-evident that confusion and complexity will multiply unchecked when one adds such facilities to an only partially understood formalism. It is simply likely to be unmanageably complicated to *attempt* to build a self-referential system unaided by the clarifying structure of a prior theory. The complexities surrounding the use of APPLY in Lisp (and the caution with which it has consequently come to be treated) bear witness to this fact. However there is a more serious problem. If one subscribes to the knowledge representation hypothesis, it becomes an integral part of developing self-descriptive systems to provide, encoded within the representational medium, an account of (roughly) the syntax, semantics, and reasoning behaviour of that formalism. In other words, if we are to build a process that "knows" about itself: and *if we subscribe to the view that knowing is representational,* then we are committed to providing that system with a *representation* of the self-knowledge with which we aim to endow it. That is, we must have an adequate theories of computational representation and reflection *explicitly formulated,* since *an encoding of that theory is mandated to play a causal role as an actual ingredient in the reflective device.*

Knowledge of any sort—and self-knowledge is no exception—is always theory relative. The representation hypothesis implies that our theories of reasoning and reflection must be explicit. We have argued that this is a substantial, if widely accepted, hypothesis. One reason to find it plausible comes from viewing the entire enterprise as an attempt to communicate our thought patterns and cognitive styles—including our reflective abilities—to these emergent machines. It may at some point be possible for understanding to be tacitly communicated between humans and system they have constructed. In the meantime, however, while we humans might make do with a rich but unarticulated understanding of computation, representation, and reflection, we must not forget that computers do not [yet[1]] share with us our tacit understanding of what they are.

[1] «The word 'yet', present in a draft written prior to submission, was for unknown reasons deleted in the very last (submitted) version.»

— Were this page been blank, that would have been unintentional —

Procedural Reflection in Programming Languages

2b — Chapter One · Introduction

The successful development of a general reflective calculus based on the knowledge representation hypothesis will depend on the prior solution of three problems:

1. The provision of a computationally tractable and epistemologically adequate descriptive language;
2. The formulation of a unified theory of computation and representation; and
3. The demonstration of how a computational system can reason effectively and consequentially about its own inference processes.

The first of these issues is the collective goal of present knowledge representation research; though much studied, it has met with only partial success. The problems involved are enormous, covering such diverse issues as adequate theories of intensionality,[x] methods of indexing and grouping representational structures, and support for variations in assertional force. In spite of its centrality, however, it will not be pursued here, in part because it is so ill-constrained.[x] The second, though it is occasionally acknowl-

[x] «Explain the use of this term—which I think I meant, at the time; rather than "intentionality." Talk about the 'intensional fusion" thesis on which Mantiq was to be based? And maybe also point to the "three spellings" sidebar in aos?»

[x] «Explain that this was, in a way, a pointer to Mantiq—which I had had to defer, in writing this dissertation—though maybe also that the project may yet see the light of day.»

edged to be important, is a much less well publicized issue, having received (so far as I know) almost no direct attention. As a consequence, every representation system proposed to date exemplifies what I will call a **dual-calculus** approach: a procedural calculus (usually LISP) is conjoined with a declarative formalism (an encoding of predicate logic, frames, etc.) into something of a formalistic hybrid. Even such purportedly unified systems as PROLOG[1] can be shown to manifest this structure. I will in passing suggest that this dual-calculus style is unnecessary and indicative of serious shortcomings in our conception of the representational endeavour. However this issue too will be largely ignored.

In this dissertation my focus instead will be on the third problem: the question of making the inferential or interpretive aspects of a computational process themselves accessible as a valid domain of reasoning. I will show how to construct a computational system whose active interpretation is controlled by structures themselves available for inspection, modification, and manipulation, in ways that allow a process to shift smoothly between dealing with a given subject or task domain, and dealing with its own reasoning processes over that domain. In computational terms, the question is one of how to construct a program able to reason about and affect its own interpretation[x]—i.e., of how to define a calculus with a reflectively accessible control structure.

1 General Overview

The term "reflection" does not name a previously well-defined question to which I propose a particular solution (although logic's *reflection principles* are not unrelated). Before I can present a theory of what reflection comes to, and how it can be demonstrated, therefore, I will have to give an account of what reflection is. In the next section, by way of introduction, I will identify six characteristics that I take to distinguish all reflective behaviour. Then,

[1] PROLOG has been presented in a variety of papers; see for example Clark and McCabe (1979), Roussel (1975), and Warren et al. (1977). The conception of logic as a programming language (with which I radically disagree) is presented in Kowalski (1974 and 1979).

[x] «Note that this is the computational notion of 'interpretation' (program execution), not the representational one familiar from logic and philosophical semantics.»

since I will be primarily concerned with **computational reflection**, I will sketch the model of computation on which the analysis will be based, and will set the general approach to reflection to be adopted into a computational context. In addition, once a working vocabulary of computational concepts has been set out, I will be able to define what I will mean by **procedural reflection**—an even smaller and more circumscribed notion than computational reflection in general. All of these preliminaries are necessary in order to enable to formulation of an attainable set of goals.

Thus prepared, I will set forth on the analysis itself. As a technical device, over the course of the dissertation I will develop three successive dialects of Lisp to serve as illustrations, and to provide a technical ground in which to work out in detail the theories of reflection to be proposed. I should say at the outset, however, that this focus on Lisp should not mislead the reader into thinking that the basic reflective architecture I propose—or the principles endorsed in its design—are in any important sense LISP specific. Lisp was chosen because it is simple, powerful, and uniquely suited for reflection in two ways: it already embodies protocols whereby programs are represented in first-class accessible (data) structures, and it is a convenient formalism in which to express its own meta-theory—especially given that I will use a variant of the λ-calculus as a mathematical meta-language (this convenience holds especially in a statically scoped dialect of the sort that will ultimately be adopted). Nevertheless, as I will discuss in the concluding chapter, it would be possible to construct a reflective dialect of Fortran, Smalltalk, or any other procedural calculus, by pursuing essentially the same approach as I will demonstrate here for Lisp.

The first Lisp dialect (called **1-Lisp**) will be an example intended to summarise current practice, primarily for comparison and pedagogical purposes. The second (2-Lisp) differs rather substantially from 1-Lisp, in that it is modified with reference to a theory of declarative denotational semantics (i.e., a theory of the denotational significance of s-expressions) formulated *independent of the behaviour of the interpreter*. The interpreter is then subsequently defined with respect to this theory of attributed[x] semantics, so that the result of processing of an expression—i.e., the value of the function computed by the basic interpretation process—is a *normal-form co-designator* of the input expression. I will call 2-Lisp a **semantically rationalised** dialect, and will argue that it makes explicit much of the understanding of Lisp that tacitly organises most programmers' understanding of Lisp but that has never been made an articulated part of Lisp theories.[x] Finally, a procedurally reflective Lisp called **3-Lisp** will be developed, semantically and structurally based on 2-Lisp, but modified so that reflective procedures are supported, as a vehicle with which to engender the sorts of procedural reflection we will by then have set as our goal. 3-Lisp differs from 2-Lisp in a variety of ways, of which the most important is the provision, at any point in the course of the computation, for a program to *reflect* and thereby obtain fully articulated "descriptions," formulated with respect to a primitively endorsed and encoded theory, of the state of the interpretation process that was in effect at the moment of reflection. In this particular case, this will mean that a 3-Lisp program will be able to access, inspect, and modify standard 3-Lisp normal-form designators of both the environment and continuation structures that were in effect a moment before.

More specifically, 1-Lisp, like Lisp 1.6 and all Lisp dialects in current use, is at heart a *first-order* language, employing metasyntactic facilities and dynamic variable scoping protocols to partially mimic higher-order functionality. Because of its metasyntactic powers (paradigmatically exemplified by the primitive QUOTE), 1-Lisp contains a variety of inchoate reflective features, all of which we will examine in some detail: support for metacircular interpreters, explicit names for the primitive processor functions (EVAL and APPLY), the ability to *mention* program fragments, protocols for expanding macros, and so on and so forth. Though I will ultimately criticize much of 1-Lisp 's structure (and its under-

[x] «Say: only thought then that it had to be attributed; explain why that was reasonable, why I didn't end up believing it, etc.)

[x] «Say: this "mechanism honouring semantics" is like derivation in logic honouring (what logic calls) interpretation. This should be clearly stated somewhere; refer to that … »

lying theory), I will document its properties in part to serve as a contrast for the subsequent dialects, and in part because, being familiar, 1-Lisp can serve as a base in which to ground the analysis.

After introducing 1-Lisp, but before attempting to construct a reflective dialect, I will subject 1-Lisp to rather thorough semantical scrutiny. This project, and the reconstruction that results, will occupy well over half the dissertation. The reason is that the analysis will require a reconstruction not only of Lisp but of computational semantics in general. I will argue in particular that it is crucial, in order to develop a comprehensible reflective calculus, to have a semantical analysis of that calculus that makes explicit the tacit attribution of significance that I will claim characterises every computational system. I take this attribution of semantical import to computational expressions to be *prior* to any account of what *happens* to those expressions: thus I will argue for an analysis of computational formulae in which **declarative import** and **procedural consequence** are independently formulated.[x] I claim, in other words, that programming languages are better understood in terms of *two* semantica1 treatments (one declarative, one procedural), rather than in terms of a single one, as is exemplified by current approaches (although interactions between them may require that these two semantical accounts be formulated in conjunction).

This semantical reconstruction is at heart a comparison and combination of the standard semantics of programming languages on the one hand, and the semantics of natural human languages and of descriptive and declarative languages such as predicate logic, the λ-calculus, and mathematics, on the other. Neither will survive intact: the approach I will ultimately adopt is not

[x] «This is what I said, but it is not strictly correct. What is intended is that the declarative import precedes (ontologically and explanatorily) the procedural consequence, and then procedural consequence (what happens to program fragments, how they are executed) is defined to *honour* that declarative import. Logically, the analytic structure would allow procedural consequence (execution) to be defined arbitrarily; in fact, the point of calling it a "semantical system" stems from the dependence that processing bears on (declarative, not procedural) interpretation.»

strictly compositional in the standard sense (although it is recursively specifiable), nor are the declarative and procedural facets entirely separate. (For example, the procedural consequence of executing a given expression may affect the subsequent context of use that determines what another expression declaratively designates.) Nor are the consequences of this approach minor. For example, I will show that the traditional notion of *evaluation*, in terms of which all Lisps to date have been defined, is both confusing and confused, and must be separated into independent notions of **reference** and **simplification**. I will be able to show, in particular, that 1-Lisp "evaluator" de-references some expressions (such meta-syntactic terms as `(QUOTE X)`, for example), and does not dereference others (such as the numerals and `T` and `NIL`). I will argue instead for what I will call a **semantically rationalised** dialect, in which the simplification and reference primitives are kept strictly distinct.

The basic thesis on which this work depends is that semantical cleanliness (along the lines suggested above) is by far the most important pre-requisite to any coherent treatment of reflection. However, as well as advocating *semantically rationalised* computational calculi, in the Lisp case I will also espouse an aesthetic I call **category alignment**, by which I mean that there should be a strict category-category correspondence across the four major axes in terms of which a computation calculus is analysed:

1. Notation,
2. Abstract structure,
3. Declarative semantics, and
4. Procedural consequence

(Category alignment is a mandate satisfied by no extant Lisp dialect.) In particular, I will insist in the dialects I design and present here: (i) that each *notational* class be parsed into a distinct *structural* class; (ii) that each structural class be treated in a uniform way by the primitive processor; (iii) that each structural class serve as the normal-form designator of each semantic class; and so forth.

Category alignment is an aesthetic with consequence. I will show that the 1-Lisp programmer (i.e., all existing Lisp programmers) must in certain situations resort to meta-syntactic machinery merely because 1-Lisp fails to satisfy this mild requirement (in particular, 1-Lisp *lists*, which are themselves a derivative class formed from some pairs and one atom, serve semantically to encode both function applications and enumerations). Though it does not have the same status as semantical hygiene, categorical elegance will also prove almost indispensable, especially from a practical point of view, in the drive towards reflection.

Once these theoretical positions have been formulated, I will be in a position to design 2-Lisp. Like Scheme and the λ-calculus, 2-Lisp is a higher-order formalism: consequently, it is statically scoped, and treats the function position of an application as a standard extensional position. 2-Lisp is of course formulated in terms of the rationalised semantics being espoused here, according to which declarative semantics must be formulated for all expressions prior to, and independent of, the specification of how they are treated by the primitive processor. Consequently—and in this way 2-Lisp is radically unlike Scheme—the 2-Lisp processor is based on a regimen of **normalisation**, according to which each expression is taken into a normal-form designator of the original expression's referent, where the notion of *normal-form* is defined in part with reference to the semantic type of the symbol's *designation*, rather than (as in the case of the λ-calculus) in terms of the further (non-) applicability of a set of syntactic reduction rules.

2-Lisp 's normal-form designators are environment independent and side-effect free; thus the concept of a *closure* can be reconstructed as a *normal-form function designator*. Since normalisation is a form of simplification, and is therefore *designation-preserving*, meta-structural expressions (terms that designate other terms in the language) are not de-referenced upon normalisation, as they are when evaluated. I therefore call the 2-Lisp processor **semantically flat**, since it stays at a semantically fixed level (although explicit referencing and de-referencing primitives—primitive operations to perform what philosophers or logicians would call semantic ascent and semantic descent—are also provided, to facilitate explicit shifts in level of designation).

3-Lisp is straightforwardly defined as an extension of 2-Lisp, with respect to an explicitly articulated procedural theory of 3-Lisp embedded in 3-Lisp structures. This embedded theory, called the **reflective model**, though superficially resembling a metacircular interpreter (as shown by a glance at the code, given in the sidebar on p ■■), is *causally connected* to the workings of the underlying calculus in critical and primitive ways. The reflective model is similar in structure to the procedural fragment of the meta-theoretic characterisation of 2-Lisp that was encoded in the λ-calculus: it is this incorporation into a system of a theory of its own operations that makes 3-Lisp, like any possible reflective system, inherently theory relative. For example, whereas *environments* and *continuations* will up until this point have been theoretical posits, mentioned only in the meta-language, as a way of explaining Lisp's behaviour, in 3-Lisp such entities move from the semantical domain of the meta-language into the semantical domain of the object language, and environment and continuation designators emerge as part of the primitive behaviour of 3-Lisp protocols.[x]

More specifically, arbitrary 3-Lisp **reflective procedures** can bind as arguments (designators of) the continuation and environment structure of the interpreter that *would have been in effect at the moment the reflective procedure was called*, had the machine been running all along in virtue of the explicit interpretation of the prior program, mediated by the reflective model. Furthermore, by constructing and/or modifying these designators, and resuming the process below, such a reflective procedure may arbitrarily control the processing of programs at the level beneath it. Because reflection may recurse arbitrarily, 3-Lisp is most simply defined as:

> *An infinite tower of* 3-Lisp *processes, each engendering the process immediately below, in virtue of running a copy of the re-*

[x] Note that it is *designators* of environments and continuations that are part of the protocol. There is a sense in which environments and continuations are themselves part of the definition of 3-Lisp, but the truth of that fact should be not taken as implying that "environment structures" and "continuation structures" are a primitive part of 3-Lisp. To speak in that way would be to fail to appreciate the importance of the declarative dimension of 3-Lisp (and 2-Lisp) semantics.

flective model.

Under such an account, the use of reflective procedures amounts to running simple procedures at arbitrary levels in this reflective hierarchy. Both a straightforward implementation and a conceptual analysis are provided to demonstrate that such a machine is nevertheless finite.

3-Lisp 's reflective levels are not unlike the levels in a typed logic or set theory, although of course each reflective level contains an omega-order untyped computational calculus essentially isomorphic to (the extensional portion of) 2-Lisp. Reflective levels, in other words, are at once stronger and more encompassing than are the order levels of traditional systems. The locus of agency in each 3-Lisp level, on the other hand, that distinguishes one computational level from the next, is a notion without precedent in logical or mathematical traditions.

The architecture of 3-Lisp allows us to unify three concepts of traditional programming languages that are typically independent (three concepts we will have explored separately in 1-Lisp):

1. The ability to support metacircular interpreters;
2. The provision of explicit names for the primitive interpretive procedures (EVAL and APPLY in standard Lisp dialects); and
3. The inclusion of procedures that access the state of the implementation (usually provided as part of a programming environment, for debugging purposes).

I will show how all such behaviours can be defined within a pure version of 3-Lisp (i.e., independent of implementation), since all aspects of the state of the 3-Lisp interpretation process are available, with sufficient reflection, as objectified entities within the 3-Lisp structural field.

The dissertation concludes by drawing back from the details of Lisp development, and showing how the techniques employed in this one particular case could be used in the construction of other reflective languages—reflective dialects of current formalisms, or other new systems built from the ground up. I will show, in particular, how this approach to reflection may be integrated with notions of data abstraction and message passing—two (related) concepts commanding considerable current attention, that might seem on the surface incompatible with the notion of a system-wide declarative semantics. Fortunately, I will be able to show that this early impression is false—that procedurally reflective *and semantically rationalised* variants on these types of languages could be readily constructed as well.

Besides the basic results on reflection, there are a variety of other lessons to be taken from the investigation, of which the integration of declarative import and procedural consequence in a unified and rationalised semantics is undoubtedly the most important. The rejection of evaluation, in favour of separate simplification and de-referencing protocols, is the major, but not the only, consequence of this revised semantical approach. The matter of category alignment, and the constant question of the proper use of metastructural machinery, while of course not formal results, are nonetheless important permeating themes. Finally, the unification of a variety of practices that until now have be treated independently—macros, metacircular interpreters, EVAL and APPLY, quotation, implementation-dependent debugging routines, and so forth—should convince the reader of one of the dissertations most important claims: procedural reflection is not a radically new idea; tentative steps in this direction have been taken in many areas of current practice. The present contribution—fully in the traditional spirit of rational reconstruction—is merely one of making explicit what we all already knew.

I conclude this brief introduction with three footnotes.

First, given the flavour of the discussion so far, the reader may be tempted to conclude that the primary emphasis of this report is on procedural, rather than on representational, concerns (an impression that will only be reinforced by a quick glance through later chapters). This impression is in part illusory; as I will explain at a number of points. these topics are pursued in a procedural context because it is *simpler* than attempting to do so in a poorly understood representational or descriptive system. All of the substantive issues, however, have their immediate counter-parts in the declarative aspects of reflection, especially when such declarative structures are integrated into a computational framework. This investigation has been carried on with the parallel declarative issues kept firmly in mind; the attribution of a declarative semantics to Lisp s-expressions will also reveal my representational bias. As I mentioned in the preface, the decision to first explore reflection in a procedural context should be taken as methodological, rather than as substantive. Furthermore, it is towards a *unified* system that I ultimate want to aim. One of the morals underlying this reconstruction is that the boundaries between these two types of calculus should ultimately be dismantled.

Second. as this last comment suggests, and as the unified treatment of semantics betrays, I consider it important to unify the theoretical vocabularies of the *declarative tradition* (logic, philosophy, and to a certain extent mathematics) with the *procedural tradition* (primarily computer science). I view the semantical approach adopted here as but a first step in that direction; as suggested in the first paragraph, a fully unified treatment remains an as-yet unattained goal. Nonetheless, I have expended some effort in the work reported here to develop and present a single semantical and conceptual position that draws on the insights and techniques of both of these disciplines.

Third and finally, as the very first paragraph of this chapter suggests, the dissertation is offered as the first step in a general investigation into the construction of *generally* reflective computational calculi to be based on more fully integrated theories of representation and computation. In spite of its reflective powers, and in spite of its declarative semantics, 3-Lisp cannot properly be called fully reflective, since 3-Lisp structures do not form a descriptive language (nor would any other procedurally reflective programming language that might be developed in the future, based on techniques set forth here, have any claim to the more general term). This is not because the 3-Lisp structures lack expressive power (although 3-Lisp has no quantificational operators, implying that even if it were viewed as a descriptive language it would remain algebraic), but rather because all 3-Lisp expressions are devoid of *assertional force*. There is, in brief, no way to *say anything* in such a formalism. One can set *x* to 3, in 3-Lisp or any other procedural (i.e., programming) language; one can test whether *x* is 3; but one cannot say *that x* is 3. Nevertheless, I contend that the insights won on the behalf of 3-Lisp will ultimately prove useful in the development of more radical, generally reflective systems. In sum, I hope to convince the reader that, although it will be of some interest on its own, 3-Lisp is only a corollary of the major theses adopted in its development.

2 The Concept of Reflection

In this section I will look more carefully at the term "reflection," both in general and in the computational case, and also specify what I would consider an acceptable theory of this phenomenon. The structure of the solution I will eventually adopt will be presented only in section 5, after discussing in section 3 the attendant model of computation on which it is based. and in section 4 the conception of computational semantics to be adopted. Before presenting any of that preparatory material, however, it helps to know where we are headed.

2a The Reflection and Representation Hypotheses

In the prologue I sketched in broad strokes some of the roles that reflection plays in general mental life. In order to focus the discussion, this section consider in more detail what I will mean by the more restricted phrase "*computational reflection*." On one reading this term might refer to a successful computational model of general reflective thinking. For example, if you were able to formulate what human reflection comes to (more precisely than I have been able to do), and were then able to construct a computational model embodying or exhibiting such behaviour, you would have some reason to claim that you had demonstrated computational reflection, in the sense of a *computational process that exhibited authentic reflective activity*. 'Computational' in this sense would mean, more or less, "computer-based."

Though I have undertaken this work with this larger goal in mind, my use of the phrase is more modest, in two important ways.

First, in this dissertation I take no stand on the question of whether computational processes are able to "think" or "reason" at all, in, as it were, their own right. Certainly it would seem that most of what we take computational systems to do is *attributed*, in a way that is radically different from the situation regarding our interpretations of the actions of other people. In particular, humans are first-class bearers of what is called **semantic originality**:[x] they themselves are able to mean, without some observer having to attribute meaning to them. Computational processes, on the other hand, are at least not yet semantically original; to the extent they can be said to mean or refer at all, they do so **derivatively**, in virtue of some human finding that a convenient description (I duck the question as to whether it is a convenient *truth* or a convenient *fiction*).[2] For example, it: as you read this, you rationally and intentionally say *"l am now reading section 2,"* you succeed in referring to this section, without the aid of attendant observers. You do so *because we define the words that way*; reference and meaning and so on are not just paradigmatically but definitionally what people do. In other words your actions are the definitional locus of reference; the rest is hypothesis, and falsifiable theory. If on the other hand I "inquire" of my home computer as to the address of a friend's farm. and it "tells me" that it is on the west coast of Scotland, the computer has not referred to Scotland in any full-blooded sense—it hasn't a clue as to what or where Scotland is. Rather. it has merely typed out an address that is probably stored in an ASCII code somewhere inside it, and *I* supply the reference relationship between that spelled word and the country in the British Isles.

The *reflection hypothesis* spelled out in the prologue, about how computational models of reflection might be constructed, embodied this cautionary stance: I said there that *in as much as a computational process can be constructed to reason at all*, it could be made to reason reflectively in a certain fashion. Thus I will take the topic of computational reflection to be restricted to those computational processes that, for similar purposes, *we find it convenient to describe as reasoning reflectively*. I do this in order to avoid the question of whether the "reflectiveness" embodied in our computational models is authentically borne, or derivatively ascribed.

[x] «Reference Dennett, Haugeland, Searle, as appropriate…»

[2] For a discussion of the semantical properties of computational systems see for example Fodor (1980), Fodor (1978), and Haugeland (1978).

The setting aside worries about semantic originality is one reduction in scope; I also adopt another. Again, in the prologue, I spoke of reflection as if it encompassed contemplative consideration not only of one's self but also of one's world (and one's place therein). While I will discuss the relationship between reflection and self-reference in more detail below, it is important to acknowledge that the focus of this investigation is almost entirely on the "selfish" part of reflection: on what it is to construct computational systems able to deal *with their own ingredient structures and operations* as explicit subject matters.

This second restriction might seem to arise for simple reasons, such as that this is an easier and better-constrained subject matter (I certainly do not consider myself in a position to postulate models of thinking about external worlds). But in fact the restriction arises for deeper reasons, again having to do with the reflection hypothesis. In the architectures I develop, I consider only *internal* or *interior* processes, able to reflect on *interior* structures, which is the only world that those internal processes conceivably can have any access to. Lisp processors (interpreters), in particular, have no access to anything except fields of s-expressions; they do not interact with the world directly, but rather in virtue of running programs, engender more complex processes that interact with the world.[x]

This "interior" sense of language processors interacts crucially with the reflection hypothesis, especially in conjunction with the representation hypothesis. Not only can we restrict to our attention to ingredient processes "reasoning about" (computing over. whatever) internal computational structures, we can restrict our attention to processes *that shift their (extensional) attention to meta-structural terms.* For consider: if it turns out that I am a computational system, consisting of an ingredient process P manipulating formal representations of my knowledge of the world, then according to the representation hypothesis, when I think, say, about Virginia Falls in northern Canada, my ingredient processor P is manipulating *representations* that are about Virginia Falls. Suppose. then, that I back off a step and comment to myself

[x] These paragraphs are awkward; and too wordy. I should compact them…

that whenever I should be writing another sentence I have a tendency instead to think about Virginia Falls. What do we suppose that my processor P is doing now? Presumably ("presumably", at least, according to the knowledge representation hypothesis. which, it is important to reiterate, we are under no compulsion to believe) my processor P is now manipulating *representations of my representations of Virginia Falls*. In other words, *because we are focused on the behaviour of interior processes,* not on compositionally constituted processes, *our exclusive focus on self-referential aspects of those processes is all we can do* (given our two governing hypotheses) *to uncover the structure of constituted, genuine reflective thought.*

The same point can be put another way. The reflection hypothesis docs not state that, in the circumstance just described, P will *reflect* on the knowledge structures representing Virginia Falls (in some weird and wondrous way)—this would be an unhappy proposal, since it would not offer any hope of an *explanation* of reflection. On pain of circularity, reflective behaviour—the subject matter to be explained—should not occur as a phenomenon in the explanation. Rather, the reflection hypothesis is at once much stronger and more tractable (although perhaps for that very reason less plausible): it posits, as an explanation of the mechanism of reflection, that the constituent interior processes compute over a *different kind of symbol.* The most important feature of the reflection hypothesis, in other words, is its tacit assumption that the computation engendering reflective reasoning, although it may be over a different kind of structure, is nonetheless *similar in kind* to the sorts of computation that regurlarly proceed over normal structures.

In sum, it is methodological allegiance to the knowledge representation hypothesis that underwrites my self-referential stance.

Though I will not discuss this meta-theoretic position further, it is crucial that it be understood, for it is only because of it that I have any right to call this inquiry a study of *reflection,* rather than a (presumably less interesting) study of *computational self-*

reference.[x]

2b Reflection in Computational Formalisms

Turn, then, to the question of what it would be to make a computational process reflective in the sense just described.

At its heart, the problem derives from the fact that in traditional computational formalisms the behaviour and state of the interpretation process are not accessible to the reasoning procedures: the interpreter forms part of the tacit background in terms of which the reasoning processes work. Plus, in the majority of programming languages, and in all representation languages, only the uninterpreted data structures lie within the reach of a program. A few languages, such as Lisp and Snobol,[x] extend this basic provision by allowing program structures to be examined, constructed, and manipulated as first class entities. What has never before been provided is a high level language in which the process that interprets those programs is also visible and subject to modification and scrutiny. Therefore such matters as whether the interpreter is using a depth-first control strategy, whether free variables are dynamically scoped, how long the current problem has been under investigation, or what caused the interpreter to start up the current procedure, remain by and large outside the realm of reference of standard representational structures. One way in which this limitation is partially overcome in some programming languages is to allow procedures access to the structures of the implementation (examples: MDL, InterLISP, ete.[3]), although such a solution is inelegant in the extreme, defeats portability and coherence, lacks generality, and in general exhibits a variety of misfea-

[x] Think about whether this is more subtle than the point in the Varieties paper, or perhaps in the annotation to the POPL paper …

[x] Snobol ("String Oriented Symbolic Language"), a string-processing language developed in the 1960s at AT&T Bell Laboratories, allowed strings to be treated as programs (programs could thus be dynamically constructed and executed on the fly). Famous for treating patterns as a first-class data type, Snobol served in some ways as a precursor to such modern languages as Perl.

[3] Such facilities as are provided in MDL are described in Galley and Pfister (1975); those in InterLISP, in Teitelman (1978).

tures that I will examine in due course. In more representational or declarative contexts no such mechanism has been demonstrated, although a need for some sort of reflective power has appeared in a variety of contexts (such as for overriding defaults, gracefully handling contradictions, etc.).

A striking example comes up in problem-solving: the issue is one of enabling simple declarative statements to be made about how the deduction operation should proceed For example, it is sometimes suggested that a *default* should be implemented by a deductive regime that accepts inferences of the following non-monotonic variety (i.e., if "not P" cannot be proved, then deduce P):

$$\frac{\neg \vdash \neg P}{P} \tag{1}$$

Though it is not difficult to build a problem solver that *embodies* such behaviour (at least on some computable reading of "not provable"). one typically does not want such a rule to be obeyed indiscriminately, independent of context or domain. On the contrary, there are usually constraints on when such inferences are appropriate—having to do with, say, how crucially the problem needs a reliable answer, or with whether other less heuristic approaches have been tried first What people writing problem-solver systems have wanted is a way to write down specific instances of something like (1) that explicitly refer both to the subject domain *and to the state of the deductive apparatus*, which, *in virtue of being written down*, lead that inference mechanism to behave in the way described.

Particular examples are easy to imagine. Thus consider a computational process designed to repair electronic circuits. One can imagine that it would be useful to have inference rules of the following sort: "*Unless you have been told that the power supply is broken. you should assume that it works*", or, "*You should make checking capacitors your first priority, since they are more likely than resistors to break down*". Furthermore, it would be good to ensure that such rules could be modularly and flexibly added and removed from the system, without each time requiring surgery on the inner constitution of the inference engine. Though we are skirting close to the edge of an infinite regress, it is clear that something like this kind of protocol is a natural part of normal human conversation. From an intuitive point of view it seems perfectly reasonable to say: *By the way, if you ever want to assume* P, *it would be sufficient to establish that you cannot prove its negation.* The question is whether we can make *formal* sense out of this intuition.

Clearly enough, the problem is not so much one of *what* to say, but of *how* to say it (to some kind of theorem-prover, for example) in a way that on the one hand does not lead to an infinite regress, and that on the other genuinely affects its behaviour. All sorts of technical question arise. It is not obvious what language to use,, for example; or even to whom such a statement should be directed. Suppose, for example, that we were supplied with a monotonic natural-deduction based theorem prover for first order logic. Could we supply it with (1) as an ordinary material implication? Certainty not. At least in the form given above, it is not even a well-formed sentence. There are various ways we could *encode* it as a sentence—one way would be to use set theory, and to talk explicitly about the set of sentences derivable from other sentences, and then to say that if the sentence '¬P' is not in a certain set, then 'P' is. The problem is that while such a sentence might contribute to a *model* of the kind of inference procedure we desire, in any ordinary theorem prover simply adding it to the stock of implication that it has to work with would not *thereby cause the inference mechanism itself behave non-monotonically in the described way.* Rather than constructing a non-monotonic reasoning system, all we would have done is to "teach" a monotonic one about non-monotonic reasoning. While such a formulation might be of interest in the specification of the constraints a reasoning system must honour (a kind of "competence theory" for non-monotonic reasoning[4]), it would not help us, at least on the face of things, with the question of how a system using defaults might actually be deployed. Another option, of course, would be to build a non-monotonic inference engine from scratch, using expressions like (1) to constrain its behaviour, along the lines of program requirements and abstract program specifications. But this would solve the problem by avoiding it—the whole question was how to use such comments on the reasoning procedure coherently *within*

[4] Reiter (1978), McDermott and Doyle (1978), Bobrow (1980).

the structures of the problem-specific application.

Yet another possibility—one I wish to focus on for a moment—is to design a more complex inference mechanism to react appropriately not only to sentences in the standard object language, but to meta-theoretic expressions of the form (1). Although no system of just this sort has been demonstrated, such a program is readily imaginable, and various dialects of PROLOG—perhaps most clearly the IC-PROLOG of Imperial College[5]—are best viewed in this light The problem with such solutions, however, is their excessive rigidity and inelegance, coupled with the fact that they do not really solve the problem in any case. What a PROLOG user is given is not a unified or reflective system, but a pair of two largely *independent* formal systems: a basic declarative language in which facts about the world can be expressed, and a *separate* procedural language, through which the behaviour of the inference process may be controlled. Although the elements of the two languages are mixed in a PROLOG program, they are best understood as separate aspects. One set (the structure of clauses, implications, and predicates, the identity of variables, and so forth) constitutes the *declarative* language, with the standard semantics of first-order logic. Another (the sequential ordering of the sentences and of the predicates in the premise, the "consumer" and "producer" annotations on the variables. the "cut" operator, and so forth) constitute the *procedural* language. Of course the flow of control is affected by the declarative aspects, but this is just like saying that the flow of control of an ALGOL program is affected by its data structures.

Thus the claim that to use PROLOG is to "program in logic" is in my view misleading: rather, what happens is that one essentially writes programs in a new (and, as it happens, rather limited) control language, using an encoding of first-order logic as the declarative representation language (i.e., as the field of data structures). Of course this is a dual system with a striking fact about its procedural component: all conclusions that can be reached are guaranteed to be valid implications of prior structures in the representational field. As mentioned above, however, this dual-calculus approach seems ultimately rather baroque, and is certainly not

[5] Clark and McCabe (1979).

conducive to the kind of reflective abilities we are after. It would be far more elegant to be able to say, *in the same language as the target world is described*, whatever it was salient to say about how the inference process was to proceed.

For example, to continue with the PROLOG example, one would like to say both FATHER(BENJAMIN,CHARLES) and CUT(CLAUSE-13) or DATA-CONSUMER(VARIABLE-4) in one and the same language, with both subject to the same semantical and procedural treatment. The increase in elegance, expressive power, and clarity of semantics that would result are too obvious to belabour: just a moment's thought leads to one realise that only a single semantical analysis would be necessary (rather than two); the reflective capabilities could recurse without limit (PROLOG and other dual-calculus systems intrinsically consist of just a single level); a meta-theoretic description of the system would have to describe only one formal language, not two; descriptions of the inference mechanism, would be immediately available, rather than having to be extracted from procedural code; and so forth.

This ability to pass coherently between two situations—in the reflective case to have the structures that normally control the interpretation process be fully and explicitly visible to (and manipulable by) the reasoning process, and in the other to allow the reasoning process to sink into them, so that they may take their natural effect as part of the tacit background in which the reasoning process works—this ability is a particular form of reflection that I will call **procedural reflection** ("procedural" because I are not yet requiring that those structures at the same time *describe* the reasoning behaviours they engender; that is the larger task not yet taken on). Although ultimately limited, in the sense that a procedurally reflective calculus is by no means a fully reflective one, even this more modest notion is on its own a considerable subject of inquiry.

2c Six General Properties of Reflection

Given the foregoing sketch of the task, it is appropriate to ask, before plunging into details, whether we can have any sense in advance of what form the solution might take. Six properties of reflective systems can be identified straight away—features that any ultimate solution should exhibit, however it ends up being structured and/or explained.

tured and/or explained.

2c.i *Causal connection*

First, the notion is one of self-reference, of a *causally-connected* kind, stronger than the notion explored by mathematicians and philosophers over much of the last century. What is needed is a theory of the causal powers required in order for a system's possession of self-descriptive and self-modelling abilities to *actually* matter to it—a requirement of substance, since full-blooded, actual behaviour is our ultimate subject matter, not simply the mathematical characterisation of formal relationships.

In dealing with computational processes, we are dealing with artefacts *behaviourally* defined, after all, unlike systems of logic, which are *functionally* defined abstractions that in no way behave or participate with us in the temporal dimension. Although any abstract machine of Turing power can provably model any other—including itself—there can be no sense in which such self-modelling is even *noticed* by the underlying machine (even if we could posit an *animus ex machina* to do the noticing). If, on the other hand, our aim is to build a computational system of substantial reflective power, we will have to build something that is affected by its ability to "think about itself." This holds no matter how accurate the self-descriptive model may be; you simply cannot afford simply to reason about yourself as disinterestedly and inconsequentially as if you were someone else.

Similar requirements of *causal connection* hold of human reflection. Suppose, for example, that after taking a spill into a river I analyse my canoeing skills and develop an account of how I would do better to lean downstream when exiting an eddy. Coming to this realisation is useful just in so far as it enables me to improve. If I merely smile in vacant pleasure at an *image* of an improved me, but then repeat my ignominious performance—*if in other words my reflective contemplations have no effect on my subsequent behaviour*—then my reflection will have been in vain. It is crucial, in other words, to make the move from description to reality. In addition, just as the *result* of reflecting has to affect *future* non-reflective behaviour, so does *prior* non-reflective behaviour have to be accessible to reflective contemplation; one must equally be capable of moving from reality to description. It would have been equally futile if, when I initially paused to reflect on the cause of my dunking, I had been unable to remember what I had been doing just before I capsized.

In sum, the relationship between reflective and non-reflective behaviour must be of a form such that both information and effect can pass back and forth between them. These requirements will impinge on the technical details of reflective calculi: we will have to strive to provide sufficient connection between reflective and non-reflective behaviour so that the right causal powers can be transferred across the boundary, without falling into the opposite difficulty of making them so interconnected that confusion results. (An example is the issue of providing continuation structures to encode control flow: we will provide *separate* continuation structures for each reflective level, to avoid unwanted interactions, but we will also have to provide a way in which a designator of the lower level continuation can be bound within the environment of the higher one, so that a reflective program can straightforwardly refer to the continuation of the process below it).The interactions between levels can grow rather complex. Suppose, to take another example, that you decide at some point in your life that whenever some type of situation arises (say, when you start behaving inappropriately in some fashion), that you will pause to calm yourself down, and to review what has happened in the past when you have let your basic tendencies proceed unchecked. The dispassionate fellow that you must now become is one that embodies, in their current and on-going being, a decision made now *at some future point to reflect*. Somehow, without acting in a self-conscious way from now until such a circumstance arises, you have to make it true that when the situation *does* arise, you will have left yourself in a state that will cause the appropriate reflection to happen *then*. By the same token, in the technical formalisms we design, we have to provide the ability to descend ("drop down") from a reflected state to a non-reflected one, having left the base level system in such a state so that, when certain situations occur in the future, the system will automatically reflect *at that point*, and thereby obtain access to the reasons that were marshalled in support of the original decision.

2c.ii *Self-knowledge*

Second, reflection has something, although just what remains to be seen, to do with self-*knowledge*, as well as with self-*reference*—and knowledge, as has often been remarked, is inherently theory-relative (in a way that pure self-reference is not). Just as one cannot interpret the world except through using the concepts and categories of a theory, one cannot reflect on one's self except in terms of the concepts and categories of a theory of self. Furthermore, as is the case in any theoretical endeavour, the phenomena under consideration under-determine the theory that accounts for them, even when all the data are to be accounted for. In the more common case, when only parts of the phenomenal field are to be treated by the theory, an even wider set of alternative theories emerge as possibilities. In other words, *when you reflect on your own behaviour, you must inevitably do so in a somewhat arbitrary theory-relative way.*

One of the mandates must be set for any reflective calculus, therefore, is that it be provided, represented in its own internal language, with an (in some appropriate sense) complete theory of how it is formed and of how it works.

Theoretical entities may be posited by this account that facilitate an explanation of behaviour, even though those entities cannot be claimed to have a theory-independent ontological existence in the behaviour being explained. 3-Lisp will be provided with a "theory" of 3-Lisp in 3-Lisp, for example, reminiscent of the metacircular interpreter demonstrated in McCarthy's original report[6] and in the reports of Sussman and Steele[7]—but causally connected in novel ways. In providing this primitively supported reflective model, I adopt a standard account, in which a number of notions commonly used to describe Lisp play a central role—such as that of an *environment*, just mentioned, and a parallel notion of a *continuation*. In spite of their familiarity, however, these have historically remained Lisp-*external* notions, being used only to describe (and model) Lisp, rather than figuring as first-class objects *internal to the language* in any direct sense. It is impossible in a *non-reflective* Lisp to define a predicate true only of environ-

[6] McCarthy et al. (1965).

[7] Sussman and Steele (1975); Steele and Sussman (1978a).

ments, since environments as such do not exist in such dialects. Because its reflective capacities are defined in terms of an environment and continuation-based theory, the notion of an environment becomes language-*internal* to 3-Lsip—with environment representing structures being passed around as first-class entities.

There are other possible Lisp theories, some of which differ substantially from the one I have chosen. For example, it is possible to replace the notion of environment altogether (note that the λ-calculus is explained without any such device). If a reflective dialect were defined in terms of this alternative theoretical account (call such a language 3'-Lisp), environments would no longer be a language internal concept. It would be likely, however, that this theory would posit other kinds of object, or other notions (such as α- and β-reduction), and in virtue of being reflective 3'-Lisp those notions would become language-internal. In order to reflect you have to use *some* theory and its associated theoretical concepts and entities.

2c.iii *Reflectivity vs. Reflexivity*

The third general point about reflection regards its name. I have deliberately chosen the term 'reflective,' as opposed to 'reflexive,' since there are various senses (other recent research reports not withstanding[8]) in which no computational process, in any sense I can understand, can succeed in narcissistically thinking *about the fact that it is at that very instant thinking about itself thinking about itself thinking*...—and so on and so on, like a transparent eye in a room full of mirrors.[x] The kind of reflecting I will consider—the

[8] Greiner and Lenat (1980), Genesereth and Lenat (1980).

[x] This is what I wrote at the time (1980), and so I have left it standing—but it is not a statement I would agree with today (2010). I still believe that there is a sensible intuition that motivating saying it about *local* reflexion—i.e., about the possibility of having "I am now thinking" refer to itself *quietly*, as it were, without invoking a Necker-cube like reverberation between one state and another (in something like the way in which non-well-founded set theory supports the notion of a one-element set having itself as its sole member). While not necessarily easy, I believe that this can be done accomplished—and, perhaps oddly but perhaps not, that doing so relates to various forms of self-referential discipline trained in various Asian meditative traditions. More seriously, unlike some others I do not believe that either

kind that 3-Lisp demonstrates how to technically define, implement, and control—requires that in the act of reflecting the process "take a step back" in order to allow the interpreted process to consider what it was just up to from a different vantage point, to bring into view symbols and structures that describe its state "just a moment earlier." From the mere fact of a system's having a name for itself it does not follow that the system thereby automatically acquires the ability to *focus on its current instantaneous self*, for in the process of "stepping back" or reflecting, the "mind's eye" moves out of its own view, being replaced by an (albeit possibly complete) account of itself. (Though this description is surely more suggestive than incisive, it is my hope that the technical work to be presented will help to allow us to make it precise.)

2c.iv *Fine-grained control*

Fourth, in virtue of reflecting a process can always obtain a finer-grained control over its behaviour than would otherwise be possible. What was previously an inexorably atomic stepping from one state to the next is opened up so that each move can be analysed, countered, and so forth—and also be broken down into constituent parts. As we will see in detail, in this way reflective powers give a system a far more subtle and more catholic—if less efficient—way of reacting to a world. The requirement here is the usual one: for what was previously implicit to be made explicit, albeit in a controlled and useful way, without violating the ultimate truth that not everything can be made explicit in a finite mechanism. This ability enables a system designer to satisfy what might otherwise be taken to be incompatible demands: (i) the provision of a small and elegant kernel calculus, with crisp defini-

the meaning or the truth of such statements as that "all statements are perspectival" need in any way be undermined by the fact that they apply, among other things, to themselves.

As explained in "Varieties of Self-Reference," «check ■■» I use 'reflexive' to refer to states, processes, expressions, etc., that include themselves within their referential or semantic extension; 'reflective,' as here, to refer to processes of "stepping back" and assaying, from a distinct vantage point, another part or aspect or period of oneself. There is no doubt that, according to this distinction, 3-Lisp was correctly described as a model of computational reflection, not or computational reflexion.

tion and strict behaviour; and at the same time (ii) the ability for the user (by using reflection) to be able to modify or adjust the behaviour of this kernel in peculiar or extenuating circumstances. One of reflection's great powers is that it allows such simplicity and flexibility to be achieved simultaneously.

2c.v *Vantage point*

This leads to the fifth general comment, which is that the ability to reflect never provides a complete separation, or an utterly objective vantage point from which to view either oneself or the world. No matter now reflective any given system or person may be, it remains a truism that there is ultimately no escape from being the person in question. Though as the dissertation proceeds I will increasingly downplay any connection between the formal work presented here and human abilities, it is still perhaps helpful to say that the kind of reflection to be presented here is closer to what is known as *detachment* or *awareness* than it is to a strict kind of self-objectivity (this is why I have been and will remain systematically imprecise about whether *reflection* is fundamentally a way to think about oneself or a way to think about the world).

The environment example just mentioned provides an illustration in a computational setting. As we will see in detail, the environment in which are bound the symbols that a program is using is, at any level, merely part of the embedding background in which the program is running. The program operates within that background, dependent on it but—in the normal (unreflective) course of events—unable to access it explicitly. The operation of reflecting makes explicit what was just implicit: it renders visible what was tacit, what was in the background. In doing so, however, a new background fills in to support the reflective deliberations. Again, the same is true of human reflection: you and I can interrupt our conversation in order to sort out the definition of a contentious term. but—as has often been remarked—we do so using other terms. Since language is our inherent medium of communication, we cannot step out of it to view it from a completely independent vantage point. Similarly, while the systems I will show how to build can at any point back up and *mention* what was previously *used*, in doing so more structured background will come into implicit use.

This lesson, of course, has been a major one in philosophy at least since Peirce; certainly Quine's famous comment about Ncurath's boat holds as true for the systems we design as it does for us designers.[9]

2c.vi *Reflectivity vs. Reflexivity*

Sixth and finally, the ability to reflect is something that must be built into the heart or kernel of a calculus. There are theoretically demonstrable reasons why reflective powers cannot be "progrrammed up" as an addition to a calculus (though one can of course *implement* a reflective machine in a non-reflective one: the difference between these two must always be kept in mind). The reason for this claim is that, as discussed in the first comment, *being reflective* is a stronger requirement on a calculus than simply *being able to model the calculus in the calculus,* something of which any machine of Turing power is capable (this is the "making it matter" that was alluded to above). This will be demonstrated in detail; the crucial difference, as suggested above, comes in connecting the self-model to the basic interpretation functions in a causal way, so that (for example and very roughly) when a process "decides to assume something," it can thereby in fact assume it, rather than simply constructing a model or self-description or hypothesis that *claims* that it is assuming it. As well as "backing up" in order to reflect on its thoughts or operations, in other words, a reflective process must be able to "drop back down again" to consider the world directly, in accord with the consequences of those reflections. Both parts of this involve a causal connection between the explicit programs and the basic workings of the abstract machine, and such connections cannot be "programmed into" a calculus that does not support them primitively.

2d Reflection and Self-Reference

At the beginning of this section I said that my investigation of reflection in general would primarily concern itself, because of operating under the knowledge representation hypothesis, with the *self-referential* aspects of reflective behaviour. There has been in the last century no lack of investigation into self-referential ex-

[9] Quine (1953a), p. 79 in the 1963 edition.

pressions in formal systems, especially since it has been exactly in these areas where the major results on paradox, incompleteness, undecidability, and so forth, have arisen. It is therefore helpful to compare the present enterprise with these theoretical precursors.

Two facets of the computational situation show how very different our concerns here will be from these more traditional studies. First, although I do not formalise this, there is no doubt in my work that I consider the locus of *referring* to be *an entire process*, not a particular expression or structure (especially not a solitary expression or structure). Even though I will posit declarative semantics for individual expressions, I will also make evident the fact that the designation of any given expression is a function not only of that expression itself, but also of the state of the processor *at the point of that expression's use*. And to the extent that "use" is even a coherent term for symbolic activity, it is the processor that uses the symbol; the symbol does not use itself. To the extent that we want a system to be self-referential, then, we want *the process as a whole* to be able to refer, to first approximation, *to its whole self*, although in fact this usually reduces to a question of it referring to some of its own ingredient structure.

Achieving this goal is not only not met by providing the system with self-referential structure, but even more strongly, I avoid such self-referential structures entirely, exactly to avoid many of the intractable (if not inscrutable) problems that arise in such cases. Because of it's λ-calculus base, it is perfectly possible in 3-Lisp to construct apparently self-designating expressions (at least up to type-equivalence: token self-reference is more difficult). But from a practical point of view the system of levels I will embrace will by and large exclude such local self-reference from our consideration. Truly self-referential expressions, such as *This sentence is six words long*, are unarguably odd, and certain instances of them, such as the clichéd *This sentence is false*, are undeniably problematic.[10] None of these truths impinge particularly on our

[10] Strictly speaking, of course, the sentence "*This sentence is six words long*" *contains* a self-reference, but is not itself self-referential. We could instead construct the composite term '*This five word noun phrase*'—though it is not as immediately evident that this leads to trouble. However the point is that the kind of reflection I am aiming for in 3-Lisp is of quite a different kind, and has no need to any such convolutions.

quite different concerns.

The second comment illustrating how different 3-Lisp and procedural are from mathematical and logical studies of self-reference is this: in traditional formal systems, the *actual reference* relationship between any given expression and its referent (whether that referent is itself or a distal object) is mediated by an externally attributed semantical interpretation function. The sentence *"This sentence is six words long"* does not actually *refer*. in any causal full-blooded sense. to anything; rather, we English speakers *take it* to refer to itself. The reference relation connecting that sentence in its role as sign, and that same sentence in its role as referent or significant, flows through us.

As emphasized in the previous section in the discussion of causal connection, in constructing reflective computational systems it is crucial for the causal mediation *not to be* be deferred through an external observer. Reflection in a computational system *has to be causally connected internally,* even if the *semantical* understanding of that causal connection is externally attributed. For example, in 3-Lisp there is a primitive relationship that holds between a certain kind of symbol, called a *handle* (a canonical form of meta-descriptive rigidly-designating name) and another symbol that, semantically, each handle designates. I.e., handles are the 3-Lisp structural form of *quotation*. Suppose that H_1 is a handle, and that S_1 is some structure that H_1 refers to. Strictly speaking, there is an internal structural relationship between H_1 and S_1, which we, as external semantical attributors, take in addition to be a *reference* relationship. Until we can construct computational systems that are what I have called semantically original. the *semantical* import of that relationship will always remain externally mediated. But the *causal* relationship between H_1 and S_1 *must be* internal: otherwise there would be no way for the internal computational processes to treat that relationship in any way that mattered.

This may be clearer if put a bit more formally. Suppose that φ is the *externally attributed* semantical interpretation function, and that ζ is the primitive *structural* function that relates handles to those structures we call their referents. It is ζ that will allow the processor to produce or obtain causal access to a structure S given that H is its handle. Thus in the prior example, it is true both that

$\varphi(\mathrm{H}_1)=\mathrm{S}_1$, due to our external semantical attribution of reference to H, and that $\zeta(\mathrm{H}_1)=\mathrm{S}_1$. More generally, we know that:

$$\forall \mathrm{H,S}\ [\![\mathrm{handle}(\mathrm{H}) \land \zeta(\mathrm{H})=\mathrm{S}]\!] \supset [\Phi(\mathrm{H})=\mathrm{S}]\!] \tag{2}$$

However, though in some sense it is strictly ~~true~~,[x] this equation in no way reveals the *structure* of the relationship between φ and ζ; it merely states their extensional equivalence. More revealing of the fact that I take the relationship between handles and referents to be a reference relation (if I may wantonly reify relationships for a moment) is the following:

$$\varphi(\zeta)=\varphi \tag{3}$$

Of, rather, since not all symbols are handles. as:

$$\varphi(\zeta) \subset \varphi \tag{4}$$

The requirement that reflection *matter*, to summarise. is a crucial facet of computational reflection—one without precedent in pre-computational formal systems. What is striking is that the mattering cannot be derived from the semantics, since it would appear that mattering—which requires a real causal connection—is a *precursor* to semantica1 originality, not something that can *follow* semantical relationships. Put another way. in the inchoately semantical computational systems I are trying to build, the reference relationships between internal meta-level symbols and their internal referents (the semantical relationships crucial in reflective considerations) may have to be causal in *two distinct ways:* once mediated by us, who attribute semantics to those symbols in the first place, and a second time internally, so that the appropriate causal behaviour, to which we attribute semantics, can be engendered. On that day when we succeed in constructing semantically original mechanisms, those two presently independent causal connections may merge; until then we will have to content ourselves with *causally original* but *semantically derivative* systems. The reflective dialects I will propose will all be of this form.

[x] It is false. «Explain»

3 A Process Reduction Model of Computation

I next want to sketch the model of computation on which the analysis and design of 3-Lisp will depend.

I take **processes** to be the fundamental subject matter; though I will not define the concept precisely, we can assume that a process consists approximately of a connected or coherent set of events through time. The reification of processes as objects in their own right—composite and causally engendered—is a distinctive, although not distinguishing, mark of computer science. Processes are inherently temporal, but not otherwise physical:[x] they do not have spatial extent, although they *must* have temporal extent Whether there are more abstract dimensions in which in is appropriate to locate a process is a question I will sidestep; since this entire characterisation is by way of background for another discussion, I will rely more on examples and on the uses to which we put these objects than on explicit formulation.

Figure 1

I will depict processes as in figure 1. The boundary of the icon is intended to signify the *boundary* or *surface* of the process itself, taken to be the interface between the process and the world in which it exists (I take objectifying a process to involve "carving them" out of a world in which it can then be said to be embedded). Thus the set of events that collectively form the behaviour of a coherent process in a given world would consist of all events on the surface of this abstract object. This set of events could be more or less specifically described: we might simply say that the process had certain gross input/output behaviour (with "input" and "output" being defined as a certain class of surface perturbation—an interesting and non-trivial problem), or we might account in tine detail for every nuance of the process's behaviour, including the exact temporal relationships between one event and

[x] At the time this was written, I was already starting to reject the claim that computational processes are *formal*, in the sense of operating independently of their semantic interpretation (in spite of what is being said in this passage), but had yet to question adequately another assumption: that computational arrangements are *abstract*. See «ref AOS».

the next, and so forth.

It is crucial to distinguish these more and less fine-grained accounts of the *surface* of a process, on the one hand—its behavioural interface or interactions with its environment—from compositional accounts of its interior, on the other. That a process has such an "interior" is again a striking assumption throughout computer science: the role of what in computer science are universally called **interpreters,** though I myself will use the term **processors**, is a striking example.[x] Suppose for instance that one were interact with a so-called "Lisp-based editor." It is standard to assume that the Lisp interpreter (processor) is an *ingredient process* within the process with which you interact: moreover, it is understood to be the locus of *anima* or *agency* inside your editor process, that in turn supplies the temporal action or activity in the editor itself. That is, of all the interior ingredients constituting the editor, only the interpreter (processor) is understood to be *active*; all other components—specifically, the "editor program" and any associated data structures—will be static or at least passive, at least at this level of abstraction. Yet the one active ingredient (interior) process *never appears as the surface of the editor*: no user interaction with the editor (via the keyboard, say) is itself directly an interaction with the Lisp processor. Rather, the Lisp processor, in conjunction with some appropriate (passive) Lisp *program*, together engender the behavioural surface with which the user interacts.

Computer science has studied a variety of such architectures—or classes of architecture; here I will briefly mention just two, but will then focus, throughout the rest of the dissertation, on just one. Every computational process, I will assume (I will take on the question of which processes we are disposed to call computational in a moment), has within it at least one other process, which, singly or collectively, supplies the animate agency of the overall constituted process.

I will call this model a **process reduction** model of computation. since at each stage of *computational reduction* a given process is reduced in terms of constituent symbols and other processes.

[x] «Reference the discussions in other papers—POPL? Prologue? I forget where this is talked about, complete with figures, etc.»

There may be more than one internal process (in what are known as *parallel* or *concurrent* processes), or there may be just a single one (known as *serial* processes). Reductions of processes that do not posit an interior process as the source of the agency I will consider to be outside the realm of computer science proper—though of course some such reduction must at some point be accounted for, if the engendered process is ever to be realized. I will view these alternatives forms of reduction—from process to, say, behaviours of physical mechanism—to fall more within physics or electronics (or perhaps computer engineering) than within computer science *per se*. What is critical is that at some stage in a series of computational reductions this leap from the domain or processes to the domain of mechanisms be taken, as for example in the explaining how the behaviour of a set of logic circuits constitutes a processor (interpreter) for the microcode of a given computer. Given this one account of what may reasonably be called the **realization** of a computational process, an entire hierarchy of processes above it may obtain indirect realisation through a series of process reductions of the above form. For example, if microcode processor interprets a set of instructions that are the program for a macro machine (say, a CPU), then a macro processor—an interpreter (processor) for the resulting "machine language" may be said to exist. Similarly, that macro machine may in turn interpret (process) a machine language program that implements SNOBOL: thus by two stages of "process composition" (i.e., the inverse of process reduction) a SNOBOL processor is also realised.

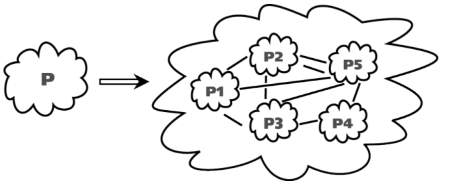


Figure 2 — Communicative Reduction

In order to make this talk of processors and so forth a little clearer, it helps to diagram two different forms of process reduction: what I will call *communicative reduction* and *interpretive reduction*. Taking the arrow '⇒' to mean "reduces to," figure 2 depicts communicative reduction, by showing that process P reduces to a set of five interior processes (P1…P5). What it is for processes to communicate I will not here say: I merely assume that those five ingredient processes interact in some fashion, so that taken as a composite unity their total behaviour is (i.e., can be "interpreted"[11] as) the behaviour of the thereby constituted process. Responsibility for the surface of the total process P is assumed to be shared in some way amongst the five ingredients. Examples of this sort of reduction may be found at any level of the computational spectrum—from metaphors of disk-controllers communicating with bus mediators communicating with central processors, to the message-passing metaphors in such Artificial Intelligence languages as ACTI and Smalltalk and so forth.[12]

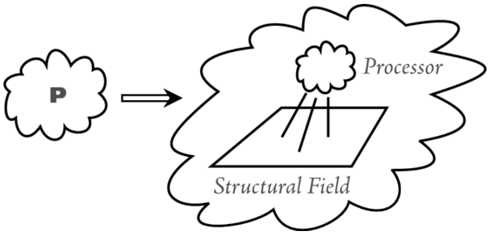


Figure 3 — Interpretive Reduction

Communicative reductions will receive only passing mention in this dissertation; I discuss them only in order to admit that the model of reflection that I will propose is not (at least at present) sufficiently general to encompass them. Instead I will focus instead on the more common model that I am calling **interpretive reduction**, pictured in figure 3.[x] In such cases the overall process is composed of what I will call a **processor** and a **structural field**. The former ingredient is the locus of active agency we have been speaking of; as already mentioned, it is what is typically called an 'interpreter;' from here on I will avoid that term (or when using it, do so within quotation marks), because of its confusion with notions of interpretation from the declarative tradition (I will have much more to say about this confusion in chapter 3).[x] The latter ingredient is intended to include both the program or the program's data structures (or both); it is often taken to consist of a set of *symbols*, although that term is so semantically loaded that for the time being I will avoid it as well.

[11] Using the English, rather than computer science, meaning of the term 'interpret.'

[12] For references on the message-passing metaphor, see Hewitt et a1. (1974) and Hewitt (1977); for ACT1 see Lieberman (1987); for Smalltalk see Goldberg (1981), Ingalls (1978).

[x] Why I did not use the phrases 'serial' and 'parallel' reduction I no longer remember; they would seem to be more appropriate terms.

[x] «Point also to other papers and commentaries as appropriate»

This second kind of reduction includes all of computer science's standard interpreted languages, of which Lisp is as good an instance as any. The Lisp structural field consists of what are known as **s-expressions**: a combination of pairs (binary graph elements of a certain form), atoms, numerals, and so forth.

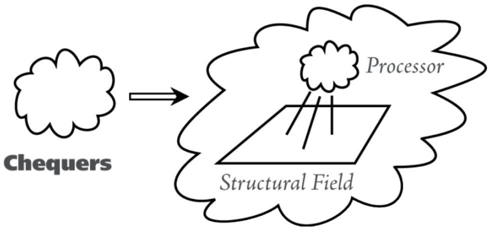


Figure 4 — First Reduction of Chequers

One benefit of the interpretive model of process reduction is that it can be used to understand both language design and the construction of particular programs.[x] For example, we can characterise Fortran in its terms, by positing a Fortran "processor" that computes over (examines, manipulates, constructs, reacts to, and so forth) elements of the Fortran structural field, which includes primarily an ordered sequence of Fortran instructions, FORMAT statements, etc. Suppose you were to set out to develop a Fortran "program" (really: process) to manage your financial affairs—which for discussion I will call Chequers. To do this, you would specify a set of Fortran data structures, and design a process to interact with them. In terms of the model, those data structures—the tables that list current balances, recent deposits, interest rates, currency conversion factors, and so on—would constitute the structural field of the first interpretive process reduction of Chequers. The "program" you design to interact with these data base I will simply call P_c. Thus the first Chequers interpretive reduction would be pictured in the model as depicted in Figure 4.

We are assuming, however, that P_c is specified by a Fortran program. P_c is not itself that program—or any program, for that matter; P_c is a *process*, and programs are static, requiring interpretation by a processor in order to engender processes or behaviour. Rather, P_c can itself be understood in terms of a second interpre-

[x] «The relation between *programs* and programming *languages* is a topic that I continue to believe is of far more theoretical importance than is normally recognized. See «…» for a discussion of the relation between programming language semantics and program semantics. … discuss … »

tive reduction, of the program C that, when processed by the Fortran processor, yields process P_c as a result. In toto, that is, the development of Chequers involves have a double interpretive reduction, depicted in Figure 5.

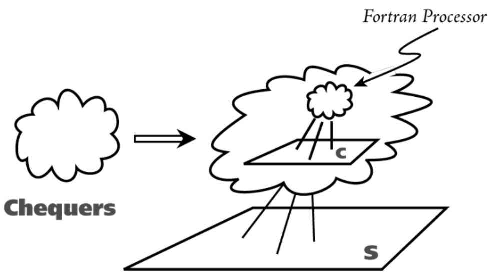


Figure 5 — Second Chequers Reduction

A host of questions would have to be answered before this model could be made precise (before, for example, one could develop anything like an adequate mathematical framework based on its underlying intuitions). For example, the data structures in the foregoing example are themselves have to be implemented in Fortran as well. However to fill out the model just a little, we can suggest how we might, in these terms, define a variety of commonplace terms of art of computer science.

First, I take it that the computer science term 'interpreter' (which, to repeat, I will call a "processor") is used in the following way:

> **Interpreter:** *A process that is the interior process in an interpretive reduction of another interior process.*

For example, the process P_c developed in the course or implementing Chequers is not interpreter, on this definition, because although it is an ingredient process (it is not, in particular, Chequers itself, but rather interior to Chequers), it is nevertheless interior only singly. The process thereby constituted—viz., Chequers—is not itself an interior process. On the other hand, it is legitimate to call the process that "interprets" (i.e., processes) Lisp programs an interpreter, because Lisp programs are structural field arrangements that engender other interior processes that work over data structures so as to yield yet other processes.

Second, I would argue that we use "compilation" as follows:

> **Compilation:** *The transformation or translation of a structural field arrangement S_1 to another structural field arrangement S_2, in such a way that the surface behaviour of the process Q_1 that*

> *would result from the processing of* S$_1$ *by some processor* P$_1$ *is equivalent—modulo some appropriate equivalence metric—to the surface behaviour of the process* Q$_2$ *that would result from the processing of* S$_2$ *by some processor* P$_2$.

For example, I spoke above about a Fortran "processor," but of course such a processor is rarely if ever realised. Rather, Fortran programs are typically compiled—usually into some form of machine language. Consider the compiler that compiles Fortran into the machine language of the IBM 360. Then the compilation of a particular Fortran program C$_F$ into an IBM 360 machine language program C$_{360}$ would be *correct* just in case the surface of the process that would result from the processing of C$_F$ by the (hypothetical) Fortran processor would be equivalent to the process that will actually result by the processing of C$_{360}$ by the basic IBM 360 machine language processor.

In sum, compilation is defined relative to two interpretive reductions, and is mandated only to ensure equivalence, modulo an appropriate metric, of resulting process surfaces.

Third, by 'implementation' I take it that we refer to two kinds of construction.

> **Process Implementation (i.e., programming):** *The construction of a structural field arrangement* S *for some processor* P *such that the surface of the process that results from the interpretation of* S *by* P *yields the desired behaviour—i.e., desired process* Q.

More interesting is to implement a **computational language**. In terms of the model, we can characterize (serial) computer languages as follows:

> **Computational Language:** *The architecture of a structural field and a behaviourally specified processor for it, in which are specified both possible arrangements or configurations of the field, and the behaviour that would result from the processing of them by the specified processor.*

In terms of this definition, we can characterize the implementation of a language:[x]

> **Language Implementation:** *The provision of a process* P *that can be interpretively reduced to the structural field and interior processor of the language being implemented.*

To implement Lisp, in other words, all that is required is the provision of a process that *behaviourally* appears to be a constituted process consisting of the Lisp structural field and the interior Lisp processor. Thus I am completely free of any actual commitment as to the reality, if any, of the implemented field.[x]

Typically, one language is implemented in another by constructing some arrangement or set of protocols on the data structures of the implementing language to encode the structural field of the implemented language. and by constructing a program in the implementing language that, when processed by the implementing language's processor, will yield a process whose surface can be taken as a processor for the interpreted language, with respect to that encoding of the implemented language's structural field. (By a *program* we refer to a structural field arrangement *within an interior processor*—i.e., to the inner structural field of a double reduction—since programs are structures that are interpreted to yield processes that in turn interact with another structural field (the data structures) so as to engender a whole constituted behaviour.)

Finally, it is straightforward to imagine how this model could be used in cognitive theorising. A **weak** computational model of some mental phenomenon or behaviour ψ would be a computational process that was claimed to be superficially equivalent to ψ (as always: modulo some equivalence metric). Note that surface equivalence of this sort can be arbitrarily fine-grained. Just because a given computational model predicts the most minute temporal nuances revealed by click-stop experiments and so forth, that does not imply that anything other than surface equivalence has been achieved In contrast, a **strong** computational model would posit not only surface but *interior architectural structure*.

[x] Is the following coherent—and correct? I am not at all sure. Tai!! …

[x] … Similarly …

Thus for example Fodor's recent claim of mental modularity[13] is a coarse-grained but strong claim: he suggests that the dominant or overarching computational reduction of the mental is closer to a communicative than to an interpretive reduction.

This has been the briefest of sketches of a substantial subject. Ultimately, it should be formalized into a generally applicable and mathematically rigorous account. In this dissertation I will merely use its basic conceptual structure to organise the analysis, and will also base the 3-Lisp architecture on it. Even for these purposes, however, it is important to identify three properties that all structural fields must manifest.

1. **Locality:** A locality metric or measure must be defined over every structural field—since (in consort with physical constraint) the interaction of a processor with a structural field is always constrained to be locally continuous.

   Informally, one can think of the processor looking at the structural field with a pencil-beam flashlight—able to see and react only to what is currently illuminated (more formally, the behaviour of the processor must always be a function only of its internal state plus the current single structural field element under investigation). Why it is that the well-known joke about a COME-FROM statement in Fortran is funny, for example,[14] can be explained only because this it violates this local accessibility constraint (it is otherwise perfectly well-defined). Note as well that in logic, the λ-calculus, and so forth, no such locality considerations come into play. In addition, the measure space yielded by this locality metric need not be uniform, as Lisp demonstrates; from the fact that A is accessible from B it does not follow that B must be accessible from A.
2. **Semantics:** it is important to the overall consideration of

[13] Fodor (forthcoming).

[14] As reported on Wikipedia, COMEFROM was initially seen in lists of joke assembly language instructions (as 'CMFRM'). It was elaborated upon in Clark, R. Lawrence, "We don't know where to GOTO if we don't know where we've COME FROM", *Datamation*, 1973, written in response to Edsger Dijkstra's "Go To Statement Considered Harmful" «ref».

semantics that structural field elements *are taken to be significant*—i.e., to be meaningful. This is why we tend to call them *symbols*. In particular, i will count as computational only those processes consisting of ingredient structures and events to which we, as external observers, attribute semantical value or import.

The reason cars are not considered to be computers, even if we treat their electronic fuel injection modules computationally, hinges on this issue of semantical attribution. The main components of a car we understand in terms of mechanics—forces, torques, plasticity, geometry, heat, combustion, and so on. These are not interpreted notions; or to put the same point another way, explaining a car does not require positing any externally attributed semantical interpretation function in order to make sense of a car's inner workings. With respect to a computer, however—whether abacus, calculator, electronic fuel injection system, or a full-scale digital computer—the best explanation is exactly in terms of the interpretation of the ingredients, even though the machine itself is not allowed access to that interpretation (for fear of violating the strictures of mechanism). Thus while I may know that the arithmetic logical unit in my machine works in such and such a way, I nevertheless "understand" its workings in terms of addition, logical operations. and so forth, all of which speak about the interpretations of its parts and workings, rather than speaking about them directly. In other words the proper use of the term "computational" is as a predicate on explanations, not on artefacts.[x]

[x] «This paragraph, and the subsequent (third) point, are clearly an informal (and not especially clear) amalgam of Fodor's "formality condition," Dennett's "intentional stance," and a distinction between original and derived intentionality. Fodor's classic formulation of the formality condition appeared in 1981 (the year this dissertation was written; see Fodor 1981); Dennett's *Intentional Stance* was not published until six years later (Dennett 1987), though formulations had appeared earlier (check■■). I no longer believe that 'computational' is best understood a predicate on explanations, though from a position that accepts derivative intentionality it

3. **Formality:** The third constraint follows directly on the second: in spite of this semantical attribution, the interior processes of a computational process must interact with these structures and symbols and other processes *in complete ignorance and disregard of any this externally-attributed semantical weight*. This is the substance of the claim that computation is *formal* symbol manipulation—that computation has to do with the interaction with symbols solely in virtue of their spelling or shape. We computer scientists are so used to this formality condition—this requirement that computation proceed syntactically—that we are liable to forget that it is a major claim, and are in danger of thinking that the simpler phrase "symbol manipulation" *means* formal symbol manipulation. Nevertheless, part of the semantical reconstruction to be undertaken here will rest on a claim that, in spite of its familiarity, we have not taken semantical attribution seriously enough.

A book should be written on all these issues; I mention them here only because they will play an important role in the upcoming reconstruction of Lisp. There are obvious parallels and connections to be explored, for example, between this external attribution of significance to the ingredients of a computational process, and the issue of what would be required far a computational system to be *semantically original* in the sense discussed at the beginning of the previous section. This is not the place for such investigations; but as §4 and chapter 3 will make clear, below, this attribution of significance to Lisp structures must be part of the full declarative semantics for Lisp. The present moral is merely that, although including such interpretation within the scope of an account of a

still does not follow that that would be so; it is a view that would deny that the property of being computational is *intrinsic*—but that is a different thing.

The main point is that, because of the fundamental thesis (that reflection is straightforward to understand and implement if built on a semantically clear base) developing this account of computational reflection and designing 3-Lisp required not only understanding such philosophical views about the nature of computing, but effectively "building them in" to the resulting reflective architecture.

language's semantics has not (to my knowledge) been done before, the attribution of semantic interpretation itself is neither something new, nor something specific to Lisp's circumstances. Externally attributed (declarative) significance is a foundational part of computer science.

4 The Rationalisation of Computational Semantics

From even the few introductory sections that have been presented so far. it is clear that semantical vocabulary will permeate the upcoming analysis. In discussing the Knowledge Representation and Reflection hypotheses, I talked of symbols that *represented* knowledge about the world, and of structures that *designated* other structures. In the model of computation just presented, I said that the attribution of *semantic significance* to the ingredients of a process was a distinguishing mark of computer science. Informally, no one could possibly understand Lisp without knowing that the atom `T` stands for truth, and `NIL` for falsity. If we subscribe to the view that computer science is about *formal* symbol manipulation, we admit not only that the subject matter involves *symbols*, but also that any computations over them must occur in ignorance of their semantical weight.[15] Even at the very highest levels, when we say that a process—human or computational—is reasoning *about* a given subject, or reasoning *about* its own thought processes, we implicate semantics, since the term 'semantics' can (at least in part) be viewed as merely a fancy word for *aboutness*.

It is therefore necessary for me to add to last section's account of processes and process reduction a corresponding accounting of the semantical assumptions I will make and techniques I will use, and to make clear what I mean when we say that I will subject computational dialects to semantical scrutiny.

4a Pre-Theoretic Assumptions

When we engage in semantical analysis, I do not take it to be our goal simply to provide a mathematically adequate specification of

[15] You cannot treat a non-semantical object, such as an eggplant or a waterfall, *formally* (unless you first, non-standardly, set it up as a symbol). The mere use of the predicate 'formal' assumes that its object is significant, or has been attributed significance, even if on the side.

the behaviour of one or more procedural calculi that would enable us, for example, to prove that programs will meet some specification of what they were designed to do. That is: by "semantics" I do not simply mean a mathematical formulation of the properties of a system, formulated from a meta-theoretic vantage point. (Unfortunately, in my view, in some writers the term seems to be acquiring this weak connotation.[x]) Rather, I take semantics to have fundamentally to do with meaning and reference and so forth—whatever they come to—as paradigmatically manifested in human thought and language (as was mentioned in §2a). I am therefore interested in semantics for two reasons: first, because, as I said at the end of the last section, all computational systems are marked by external semantical attribution; and second, because semantics is the study that will reveal what a computational system is reasoning *about,* and a theory of what a computational process is reasoning about is a pre-requisite to a proper characterisation of reflection.

Given this agenda, I will approach the semantical study of computational systems with a rather precise set of guidelines. In particular, I will require that any subsequent semantical analyses answer to the following two requirements, emerging from the two facts about processes and structural fields laid out at the end of section:

1. They should manifest the fact that we understand computational structures in virtue of attributing to them semantical import;
2. They should make evident that, in spite of such attribution, computational processes are *formal,* in that they must be defined over structures independent of their semantical weight.

[x] As explained in the annotation to the "Reflection and Semantics in Lisp" paper presented at the Principles of Programming Languages conference in 1984 (included in this volume—see ■■), at the time this dissertation was written I was in the grip of an "ingrediential" view of programs, rather than a "specificational" one, and so had not considered the position, much more commonly held in computer science, that a program was a *specification of*, rather than an ingredient within, a computational process.

These two principles alone entail the requirement of a double semantics, since the attributed semantics mentioned in the first premise includes not only a pre-theoretic understanding of *what happens* to computational symbols, but also a *pre-computational* intuition as to what those symbols *stand for*. It follows that we will have to make clear the *declarative* semantics of the elements of (in our case) the Lisp structural field, as well as establishing their *procedural* import

I will explore these results in more detail below, but in bare outlines the argument is straightforward. Most of the results are consequences of the following basic tenet (relativised here to Lisp, for perspicuity, but the same would hold for any other calculus):

> *What Lisp structures mean ;s not a function of how they are treated by the Lisp processor. Rather, how they are treated is a function of what they mean.*

For example, I take it that the Lisp expression "`(+ 2 3)`" evaluates to "`6`" for the undeniable reason that "`(+ 2 3)`" is understood as a complex *name* of the number that is the successor of four. We arrange things—we define Lisp in the way that we do—so that the numeral `6` is the value *because we know in advance what* `(+ 2 3)` *stands for*. To borrow a phrase from Barwise and Perry, this reconstruction is an attempt to "*regain our semantic innocence*"[16]—an innocence that still permeates traditional formal systems (logic, the λ-calculus, and so forth), but that has been lost in the attempt to characterise the so-called "semantics" of computer programming languages.

That "`(+ 2 3)`" designates the number five is self-evident, as are many other examples on which I will begin to erect my denotational account. I have also already alluded to the equally unarguable fact that (at least in certain contexts) `T` and `NIL` designate Truth and Falsity. Similarly, it is commonplace use the term "CAR" as a *descriptive function* to designate the first element of a pair, as for example in the English sentence "I noticed that the CAR of that list is the atom L." The important point is that, in that English sentence, the phrase "CAR of that list" occurs as a *name* or a *designator*—not as a *procedure call*. Nothing *happens*, when I say

[16] «Ref Situations and Attitudes, probably—check»

it; it is not *executed*. It is merely a way of *pointing* to something—to the first element of the list pointed to by the ingredient phrase 'that list.' Similarly, it is hard to imagine an argument against the idea that "(QUOTE X)" designates X—in contrast to the claim, which is also often heard, that does not speak at all about naming or designation, but only about procedural treatment: that QUOTE is a function that *holds off the evaluator*.

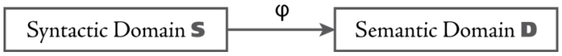


Figure 6 — Minimal Semantics

In sum, the moral is not so much that formulating the declarative semantics of a computational formalism is difficult, as that it must be recognized as an important thing to do.

2b Semantics in a Computational Setting

In the most general form that I will use the term *semantics,*[17] a semantical investigation aims to characterise the relationship between a **syntactic** domain and a **semantic** domain—a relationship typically studied as a mathematical function mapping elements of the first domain into elements of the second. I will call such a function an **interpretation** function (it was in order to be able to talk about this function, which must be sharply distinguished from what is called an 'interpreter' in computer science, that I switched to the term *processor*). Schematically, that it, as shown in figure 6, the function φ is taken to be an interpretation function from S to D.

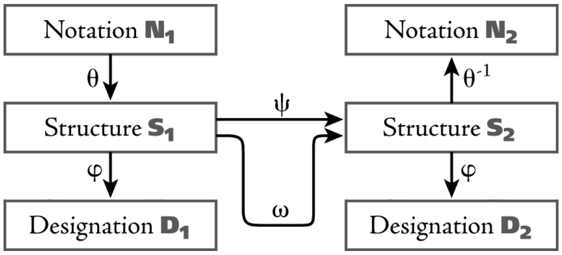


Figure 7 — Computational Semantics

In a computational setting, this simple situation is made more complex because we are studying a variety of interacting interpretation functions. In particular, figure 7 identifies the relationships between the three main semantical functions that will

[17] See the postscript, however, where I in part disavow this fractured notion of syntactic and semantic domains.

permeate the analysis of 3-Lisp. θ is the interpretation function mapping notations into elements of the structural field, φ is the interpretation function making explicit our attributed semantics to structural field elements, and ψ is the function formally computed by the language processor. ω will be explained below; it is intended to indicate a φ-semantic characterisation of the relationship between S_1 and S_2, whereas ψ indicates the formally computed relationship—a distinction similar, as I will soon argue, to that between the logical relationships of *derivability* ($\vdash$) and *entailment* ($\models$).

The names have been chosen for connotative convenience: 'ψ' by analogy with *psychology*, since it is a study of the internal relationships between and among symbols, all within the machine ('ψ' in this sense is meant to signify psychology *narrowly construed*, in the sense of Fodor, Putnam, and others[18]). The label 'φ', on the other hand, chosen to suggest *philosophy*, signifies the relationship between a set of symbols and the world. By analogy, suppose we were to accept the hypothesis that people represent or encode English sentences in an internal mental language called mentalese (suppose, in other words, that we accept the hypothesis that our minds are computational processes). If you say to me "A composer who died in 1750" and I respond with "Johan Sebastian Bach", then, in terms of the figure, the first phrase, *qua* sentence of English, would be N_1; it would "notate" or "express" the mentalese structure N_1, and the person who lived in the seventeenth and eighteenth centuries would be the referent D_1. Similarly, my reply would be N_2, the mentalese fragment that I thereby express would be S_2, and D_2 would again be the long-dead composer. I.e., in this case D_1 and D_2 would be identical.

N_1, S_1, D_1, N_2, S_2, and D_2, in other words, need not necessarily all be distinct; in a variety of different circumstances two or more of them may be one and the same entity. I will examine cases, for example, of self-referential designators, where S_1 and D_1 are the same object. Similarly, if, on hearing the phrase "the pseudonym of Samuel Clemens," I reply "Mark Twain", then D_1 and N_2 are identical. By far the most common situation, however, will be as in the Bach example, where D_1 and D_2 are the same entity—a cir-

[18] Fodor (1980).

cumstance in which I will say that the function ψ is **designation-preserving**. As we will see in the next section, the α-reduction and β-reduction of the λ-calculus, and the derivability relationship (⊢) of logic, are both designation-preserving relationships. Similarly, the 2-Lisp and 3-Lisp processors I present will be designation-preserving, whereas 1-Lisp 's and Scheme's evaluation protocols, as we have already indicated, are not.

In the terms of this figure, the argument I will present in chapter 3 will run roughly as follows. First I will review both logic systems and the λ-calculus, to illustrate the general properties of the φ and ψ employed in those formalisms, for comparison. Next I will shift towards computational systems, beginning with PROLOG, since it has evident connections to both declarative and procedural traditions. Finally I will take up Lisp. I will argue that it is not only coherent, but in fact natural, to define a declarative φ for Lisp, as well as a procedural ψ. I will also sketch some of the mathematical characterisation of these two interpretation functions. It will be clear that though similar in certain ways, they are nonetheless crucially distinct. In particular, I will be able to show that 1-Lisp 's ψ (EVAL) obeys the following equation. I will say that any system that satisfies this equation has the **evaluation property**, and the statement that, for example, the equation holds of 1-Lisp the **evaluation theorem**. (The formulation used here is simplified for perspicuity, ignoring contextual relativisation; Σ is the set of structural field elements.)

$$\forall\, S \in \Sigma\ [\ \text{if } \varphi(S) \in \Sigma \ \text{ then } \psi(S) = \varphi(S) \quad \text{else } \varphi(\psi(S)) = \varphi(S)\ ] \tag{5}$$

1-Lisp 's evaluator, in other words, **de-references** *just those structures whose referents lie within the structural field,* and is designation-preserving otherwise. Where it can, in other words, 1-Lisp 's ψ (i.e, its processor) implements φ; when it cannot, ψ is *φ-preserving,* although what it does do with its argument in this case has yet to be explained (saying that it preserves φ is too easy: the identity function preserves designation was well, but EVAL is not the identity function).

The behaviour described in (5) is unfortunate, in part because the question of whether φ(S)∈Σ is not in general decidable, and therefore even if one knows of two expressions S_1 and S_2 that S_2 is

$\psi(S_1)$, one still does not necessarily know the relationships between $\varphi(S_1)$ and $\varphi(S_2)$. More seriously, it makes the explicit use of meta-structural facilities extraordinarily awkward, thus defeating attempts to engender reflection. I will argue instead for a dialect described by the following alternative (again in skeletal form):

$$\forall S \in \Sigma\ [[\varphi(\psi(S))=\varphi(S)] \wedge [\text{NORMAL-FORM}(\psi(S))]] \qquad (6)$$

When I prove it for 2-Lisp, I will call this equation the **normalisation theorem**; I will say that any system satisfying it has the **normalisation property**. Diagrammatically. the circumstance it describes is pictured in figure 8. Such a ψ, in other words, is *always* φ-preserving. In addition, it relies on a notion of of normal-formedness, which we will have to define.

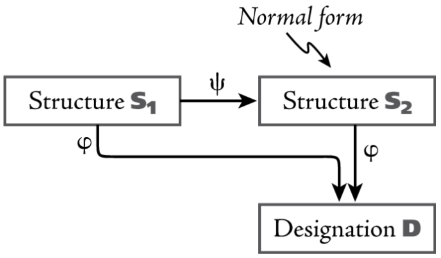


Figure 8 — Normalisation

In the λ-calculus, $\psi(S)$ would *definitionally* be in normal-form, since in that calculus normal-formedness is *defined* in terms of the non-applicability of any further β-reductions. As I will argue in more detail in chapter 3, this makes the notion less than ideally useful: in designing 2-Lisp and 3-Lisp, therefore, I will in contrast define normal-formedness in terms of the following three (provably independent) properties:

1. Normal-form designators must be **context-independent**, in the sense of having the same declarative and procedural import independent of their context of use;
2. They must also be **side-effect free**, implying that any (further) procedural treatment of them will have no affect on the structural field or state of the processor; and
3. They must be **stable**, meaning that they normalise to themselves in all contexts.

It will then require a proof that all 2-Lisp and 3-Lisp results (all expressions $\psi(S)$ are in normal-form. In addition, from the third (stability) property, plus this proof that ψ's range includes only normal-form expressions, it will be possible to show that ψ is *idempotent,* as was suggested earlier ($\psi=\psi\circ\psi$—i.e., $\forall S$

$\psi(S)=\psi(\psi(S))$—a property of 2-Lisp and 3-Lisp that will ultimately be shown to have substantial practical benefits.

There is another property of normal-form designators in 2-Lisp and 3-Lisp, beyond the three requirements just listed, that follows from the category alignment mandate. In designing those dialects I will insist that the *structural category* of each normal form designator be determinable from *the type of object designated*, independent of the structural type of the original designator, and independent as well of any of the machinery involved in implementing ψ (this is in distinction to the received notion of normal form employed in the λ-calculus, as will be examined in a moment). For example, I will be able to demonstrate that any term that designates a number will be taken by ψ into a numeral, since numerals will be defined as the normal-form designators of numbers. In other words. from just the designation of a structure S the *structural category* of $\psi(S)$ will be predictable, independent of the form of S itself (although the *token identity* of $\psi(S)$ cannot be predicted on such information alone, since normal-form designators are not necessarily unique or canonical). This category result, however, will also need to be proved: i call it the **semantical type theorem**.

That normal form designators cannot be canonical arises, of course, from computability considerations: one cannot decide in general whether two expressions designate the same function, and therefore if normal-form function designators were required to be unique, it would follow that expressions that designated functions could not necessarily be normalized. Instead of pursuing that approach, however, which I would view as unhelpful, I will instead adopt a non-unique notion of normal-form function designator, which still satisfies the three requirements specified above; such a designator will by definition be called a **closure**. All well-formed function-designating expressions, on this scheme, will succumb to a standard normalisation.

Some 2-Lisp (and 3-Lisp) examples will illustrate all of these points. I assume that the numbers are included in the semantical domain, a syntactic class of **numerals** are taken to be normal-form number designators. The numerals are canonical (one per number), and as usual are side-effect free and context-independent; thus they satisfy the requirements on normal-formedness. The semantical type theorem says that any term that designates a number will normalise to a numeral: thus if X designates five and Y designates six, and if + designates the addition function, then we know (can prove) that (+ X Y) designates eleven and will normalise to the numeral 11. Similarly, there are two boolean constants $T and $F that are normal-form designators of Truth and Falsity, respectively, and a canonical set of rigid structure designators called **handles** that are normal-form designators of all s-expressions (including themselves). And so on; closures are normal-form function designators, as mentioned above; I will also specify normal-form designators for sequences and other types of mathematical objects included in the semantical domain.

I have diverted the discussion away from general semantics, onto the particulars of 2-Lisp and 3-Lisp in order to illustrate how the semantical reconstruction I endorse impinges on language design. However, it is important to recognise that the behaviour mandated by (6) is not *new:* this is how all standard semantical treatments of the λ-calculus proceed, and the designation-preserving aspect of it is approximately true of the inference procedures in logical systems as well, as we will see in detail in chapter 3. Neither the λ-calculus reduction protocols, in other words, nor any of the typical inference rules one encounters in mathematical or philosophical logics, *de-reference* the expressions over which they are defined. In fact it is hard to imagine *defending* equation (5). Rather, it seems reasonable to speculate that because Lisp includes its syntactic domain within the semantic domain—i.e., because Lisp has QUOTE as a primitive "operation"—a semantic inelegance was inadvertently introduced into the design of the language that has never been corrected. If this is right, then the proposed rationalisation of Lisp can be understood as an attempt to *regain the semantical clarity* of predicate logic and the λ-calculus, achieved in part by connecting the language of the computational calculi with the language in which prior linguistic systems have been studied.

It is this regained coherence that I am claiming is a necessary prerequisite to a coherent treatment of reflection.

One final comment The consonance of (6) with standard semantical treatments of the λ-calculus, and the comments just made about Lisp's inclusion of QUOTE, suggest that one way to view the present project is as a semantical analysis of a variant of the λ-calculus with quotation. In the Lisp dialects I consider, I will retain sufficient machinery to handle side effects, but it is of course always possible to remove such facilities from a calculus. Similarly, we could remove the numerals and atomic function designators (i.e., the ability to name composite expressions as unities). What would emerge would be a semantics for a deviant λ-calculus with some operator like QUOTE included as a primitive syntactic construct—a semantics for a *meta-structural* extension of the already *higher-order* λ-calculus. I will not pursue this line of attack further in this dissertation, but, once the mathematical analysis of 2-Lisp is in place, such an analysis should emerge as a straightforward corollary.

4c Recursive and Compositional Formulations

If the previous sections have briefly suggested the work that I would like the proposed semantics to do, they do not reveal how this is to be accomplished. In chapter 3, where the reconstruction of semantics is laid out, I will of course pursue this latter question in detail, but I can summarise some of its results here.

Beginning very simply, standard approaches suffice. For example, I begin with *declarative import* (φ), and initially posit the designation of each primitive object type (saying for instance that the numerals designate the numbers, and that the primitively recognised closures designate a certain set of functions, and so forth), and then specify recursive rules that show how the designation of each composite expression emerges from the designation of its ingredients. Similarly, in parallel fashion I specify the *procedural consequence* (ψ) of each primitive type (saying in particular that the numerals and booleans are *self-evaluating*, that atoms evaluate to their bindings, and so forth),[x] and then once again specify recursive rules showing how the value or *result* of a composite expression is formed from the results of processing its constituents.

[x] «Check whether the two instances of 'evaluate' in that sense should be 'normalise'. Or am I still talking about 1-Lisp ?»

If we were considering only purely extensional, side-effect free, functional languages, the story might end there. However, of a variety of complications that will demand resolution, two may be mentioned here. First, none of the Lisp's that I will consider are purely extensional: there are intensional constructs of various sorts (QUOTE, for example, and even LAMBDA, which I will view as a standard intensional procedure, rather than as a syntactic mark). The hyper-intensional QUOTE operator is not in itself difficult to deal with, although I will also consider questions about the less fine-grained intensionality manifested by a statically-scoped LAMBDA. As in any system, the ability to deal with intensional constructs requires a reformulation of the semantics of all expressions—i.e., requires recasting the semantics of extensional procedures as well, in appropriate ways. This is a minor complexity, but no particular difficulty emerges.

The second difficulty has to do with side-effects and contexts. All standard model-theoretic techniques allow for the general fact that the semantical import of a term may depend in part of on the context in which it is used, of course (variables are the classic simple example). However, side-effects—which are part of the total *procedural consequence* of an expression, impinge on the appropriate context *for declarative purposes* as well as well as for procedural ones. For example, in a context in which X is bound to the numeral 3 and Y is bound to the numeral 4, it is straightforward to say that the term (+ Y Y) designates the number seven, and returns the numeral Y. However consider the semantics of the more complex (this is standard Lisp):

```
(+ 3 (PROG (SETQ Y 14) Y))                (7)
```

It would be hopeless—to say nothing of false—to have the formulation of declarative import ignore procedural consequence, and claim that (7) designates seven, even though it patently returns the numeral 17.[19] On the other hand, to *include* the procedural effect of the SETQ within the specification of φ would seem to violate the ground intuition arguing that the designation of this

[19] I say this in spite of the fact that I am under no absolute obligation to make the declarative and procedural stories cohere—in fact I will reject 1-Lisp exactly because they do *not* cohere in any way that I can accept.

term, and the structure to which it evaluates, are different.

The approach I will ultimately adopt is one in which I define what I call a **general significance function** Σ which embodies both declarative import (designation), local procedural consequence (what an expression "evaluates to," to use 1-Lisp jargon), and full procedural consequence (the complete contextual effects of an expression, including side-effects to the environment, modifications to the structural field, and so forth). Only the total significance of the dialects I define will be strictly *compositional;* the components of that total significance, such as the designation, will be *recursively specified* in terms of the designation of the constituents, relativized to the total context of use specified by the encompassing general significance function. In this way I will be able to formulate precisely the intuition that (7) designates seventeen, as well as returning the corresponding numeral 17.

Lest it seem that by handling these complexities we have lost any incisive power in the approach, I should note that it is not always the case that the processing of a term results in the obvious (i.e., normal-form) designator of its referent For example, I will prove that, in traditional Lisps, the expression

```
(CAR '(A B C))                                   (8)
```

both designates *and* returns the atom A. Just from the contrast between these two examples ((7)and (8)) it is clear that traditional Lisp processing and Lisp designation do not track each other in any trivially systematic way.

Although this approach will be shown successful, I will ultimately abandon the strategy of characterising the full semantics of standard Lisp (as exemplified in my 1-Lisp dialect), since the confusion about the semantic import of evaluation will in the end make it virtually impossible to say anything coherent about designation. This, after all, is my goal: to *judge* 1-Lisp, not merely to *characterise* it. By the time I wrap up its semantical analysis, I will have shown not only *that* Lisp is confusing, but also (in detail) *why* it is confusing—giving us adequately preparation to design a dialect that corrects its errors.

4d The Role of a Declarative Semantics

One brief final point about this double semantics.

It should be clear that it is impossible to specify a normalising processor without a pre-computational theory of semantics.[x] If you do not have an account of what structures mean, *independent of* and how they are treated by the processor, there is no way to say anything substantial about the semantical import of the function that the processor computes.[x] On the standard approach, for example, it is impossible to say that the processor is *correct*, or *semantically coherent*, or *semantically incoherent*, or any such thing; it would merely be what it is. Given some account of what it does, one can compare this to *other* accounts: thus it would for example be possible to prove that a *specification* of it was correct, or that an *implementation* of it was correct, or that it had certain other independently definable properties (such as that it always terminated, that it used certain resources in certain fashion, etc.). In addition, *given* such an account, one could prove properties of programs written in the resulting language—thus, from a mathematical specification of the processor of ALGOL, plus the listing of an ALGOL program, it might be possible to prove that that program met some specification (such as that it sorted its input, or whatever). But all of these things are compatible with the system being a purely mechanical system—such as a device that sorted apples into different bins, or for that matter was a care. However none of these questions are the question I am trying to answer here—namely: *what is the semantical character of the processor itself?*

In the particular case I am considering, I will be able to specify the semantical import of the function computed by Lisp's evaluation regimen (i.e., by EVAL—this is content of the evaluation theorem), but only by first laying out both declarative and procedural theories of Lisp. Again, I will be able to design 2-Lisp only with

[x] This is the equivalent, in a computational context, of saying something that would be obvious, logically: that one cannot specify a proof procedure (⊢) without first having in mind an interpretation function for it to honour.

[x] This is too strongly stated. Full independence is not required; the two could be co-constituted. What is true about the point made in the text is that defining a processing regimen in a calculus in which there was *nothing more* to meaning than "how the symbol or structure was treated" would not just evacuate the system of any semantic or intentional (or computational!) interest; it would deprive it of any claim to *being* a computational system. I.e., it would reduce it to nothing but pure mechanism.

reference to this pre-computational theory of declarative semantics. It is a simple point, which I am perhaps repeating too often, but it is important to make clear how the semantical reconstruction I am endorsing is a *prerequisite* to the design of 2-Lisp and 3-Lisp, not a post-facto method of analysing them.

5 Procedural Reflection

Now that we have assembled a minimal vocabulary with which to talk about computational processes and matters of semantics, it is possible to sketch the architecture of reflection that I will present in the final chapter of the dissertation.

I will start rather abstractly, with the general sense of reflection sketched in section 2, and then make use of both the Knowledge Representation Hypothesis and the Reflection Hypothesis to define a more restricted goal. Next, I will employ the characterizations of interpretively reduced computational processes and of computational semantics to narrow this goal even further. At each step in this progressive focusing process, it will become increasingly clear what would be be involved in actually constructing an authentically reflective computational language. By the end of this section I will be able to suggest the particular structure that, in chapter 5, will be embody in the 3-Lisp design.

5a A First Sketch

Begin very simply. At the outset, I characterised reflection in terms of a process shifting between a pattern of reasoning about some subject matter, world, or task domain, to reasoning reflectively about its thoughts and actions in that world. I said in the Knowledge Representation Hypothesis that the only current candidate architecture for a process that reasons *at all* (even derivatively) is one constituted in terms of an interior process manipulating representations of the appropriate knowledge of that domain. We can see in terms of the process reduction model of computation a little more clearly what this means. For the process we called Chequers to reason about the world of finance, I suggested that it be *interpretively composed* of an ingredient process P manipulating a structural field S consisting of representations of cheque books, credit and debit entices, currency exchange rates, and so forth. Thus we were led to the image depicted in figure 4 (reproduced here as figure 9).

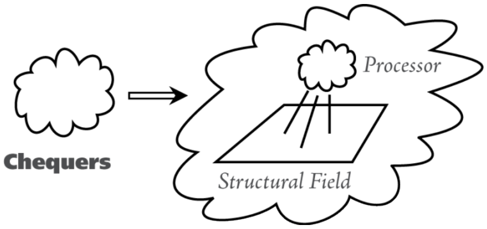


Figure 9 — First Interpretive Reduction

Next, I said (in the Reflection Hypothesis) that the only suggestion we have as to how to make Chequers reflective is this: as well as constructing process P to deal with these various financial records, we could also construct process Q to deal with P *and the structural field that* P *manipulates.* Thus Q might specify what to do when P failed or encountered an unexpected situation, based on what parts of P had worked correctly and what state P was in when the failure occurred, and so on. Alternatively, Q might describe or generate parts of P that had not been fully or adequately specified. Finally, Q might bring into existence a more complex interpretation process for P, or one particularized to suit specific circumstances. In general, whereas the world of P—the domain that P models, simulates, reasons about—is the world of finance, the world of Q is the world of the process P and the structural field it computes over.[x]

I have spoken as if Q were a *different* process from P, but whether it is really different from P, or whether it is P in a different guise, or P at a different time, is a question I will defer for a while (in part because I have said nothing about individuation criteria on processes). All that matters for the moment is that there be *some* process that does what I have said that Q must do.

What is required, in order for Q to reason about P? Because Q, like all the processes we are considering, is assumed to be interpretively composed, what is needed is what is always needed: *structural representations of the relevant facts about* P. What would such representations be like? First, they must be expressions (statements), formulated with respect to some theory, describing or representing the state of process P (we can begin to see how the *theory relative* mandate on reflection from §2 is making itself evi-

[x] That last ¶ isn't stated right; it is off one level of designation. I must fix it…

dent). Second, in order to actually describe P, they must be *causally connected* to P in some appropriate way (another of the general requirements). Thus we are considering a situation such as that depicted in figure 10, where the field (or field fragment) Sp contains these causally connected structural descriptions.

Figure 10 is of course incomplete, in that it does not suggest how Sp should relate to P (answering this question is our current quest). Note however that reflection must be able to recurse, implying the additional possibility of something like the image depicted in figure 11.

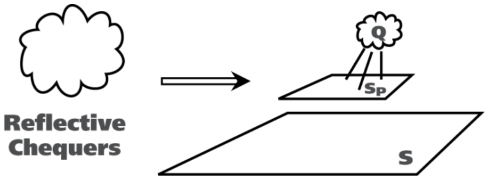


Figure 10 — Reflective Chequers, Step 1

Where might an encodable procedural theory come from? There are two possible sources: in the semantical reconstruction to be undertaken presently (before 3-Lisp is designed) I will have presented a full theory of the (non-reflective versions of the) dialects under development; this is one candidate source for an appropriate theory. But given that for the moment we are considering only *procedural* reflection, we need only the (simpler) procedural component of that theory.[20]

The second source of a theoretical account, quite similar in structure but even closer to the one we will adopt, is what we will call the **metacircular processor**, which is worth a brief examination.

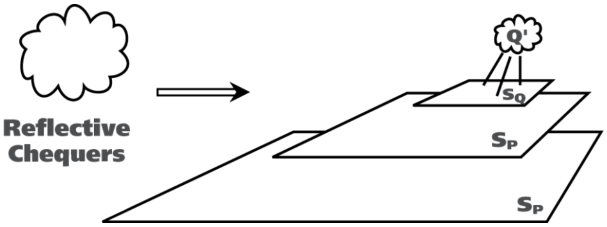


Figure 11 — Reflective Chequers, Step 2

[20] In the general case, we would need to encode, both declarative and procedurally, the full theory of computational significance.

5b Metacircular Processors

In any computational formalism in which programs are accessible as first class structural fragments, it is possible to construct what are commonly known as *metacircular interpreters:* "meta" because they operate on (and therefore terms within them designate) other formal structures, and "circular" because they do not constitute a definition of the processor, for two reasons: (i) they have to be run by that processor in order to yield any sort of behaviour (since they are *programs,* not *processors,* strictly speaking); and (ii) the behaviour they would thereby engender can be known only if one knows beforehand what the processor does. Nonetheless, such processors are often pedagogically illuminating, and they wilt play a critical role in our development of the reflective model. In line with my general strategy of reserving the word "interpret" for the semantical interpretation function. I will henceforth call such processors **metacircular processors**.

In the presentation of 1-Lisp and 2-Lisp I will construct metacircular processors (MCPs); the 2-Lisp version is presented in figure 12 (details will be explained in chapter 4; at the moment I mean only to illustrate the general structure of this code). The basic idea is that if this code were processed by the primitive 2-Lisp processor. the process that would thereby be engendered would be behaviourally equivalent to that of the primitive processor itself. In other words, if we were mathematically to take processes as functions from structure onto behaviour, and if we name the processor presented in figure 12 MCP_{2L}, and the primitive 2-Lisp processor P_{2L}, then if we taken '≅' to mean behaviourally equivalent, then we should be able to prove the following, in some appropriate sense (this is the sort of proof of correctness one finds in for example Gordon[21]):

$$\text{P}_{2L}(\text{MCP}_{2L}) \cong \text{P}_{2L} \qquad (9)$$

It should be recognised that the equivalence spoken of here is a global equivalence; by and large the primitive processor, and the processor resulting from the explicit running of the MCP, cannot be arbitrarily mixed (as already mentioned, and as a more detailed

[21] Gordon (1973 and 1975).

```
(DEFINE NORMALISE
   (LAMBDA EXPR [EXP ENV CONT]
      (COND [(NORMAL EXP) (CONT EXP)]
         [(ATOM EXP) (CONT (BINDING EXP ENV))]
         [(RAIL EXP) (NORMALISE-RAIL EXP ENV CONT)]
         [(PAIR EXP) (REDUCE (CAR EXP) (CDR EXP) ENV CONT)])))

(DEFINE REDUCE
   (LAMBDA EXPR [PROC ARGS ENV CONT]
      (NORMALISE PROC ENV
         (LAMBDA EXPR [PROC!]
            (SELECTQ (PROCEDURE-TYPE PROC!)
               [IMPR (IF (PRIMITIVE PROCI)
                         (REDUCE-IMPR PROC! ARGS ENV CONT)
                         (EXPAND-CLOSURE PROC! ARGS CONT))]
               [EXPR (NORMALISE ARGS ENV
                        (LAMBDA EXPR [ARGS!]
                           (IF (PRIMITIVE PROC!)
                               (REDUCE-EXPR PROC! ARGS! ENV CONT)
                               (EXPAND-CLOSURE PROC! ARGS! CONT))))]
               [MACRO (EXPAND-CLOSURE PROC! ARGS
                          (LAMBDA EXPR [RESULT]
                              (NORMALISE RESULT ENV CONT)))])))))

(DEFINE EXPAND-CLOSURE
   (LAMBDA EXPR [CLOSURE ARGS CONT]
      (NORMALISE (BODY CLOSURE)
                 (BIND (PATTERN CLOSURE) ARGS (ENV CLOSURE))
                 CONT)))
```

Figure 12 — A Metacircular Processor for 2-Lisp

discussion in chapter 5 will formalize). For example, if a variable is bound by the underlying processor P_{2L} it will not be able to be looked up by the metacircular code. Similarly, if the metacircular processor encounters a control structure primitive, such as a THROW or a QUIT, it will not cause the metacircular processor itself to exit prematurely, or to terminate. The point, rather, is that if an entire computation is mediated by the explicit processing of the MCP, then the results will be the same as if that entire computation had been carried out directly.

We can merge these results about MCPs in general with the diagram in figure 9 as follows: if we replaced P in the figure with a process that resulted from P processing the metacircular processor MCP (for the appropriate language—in this case assumed to be Fortran), we would still correctly engender the behaviour of Chequers, as depicted in figure 13. Furthermore, this replacement could also recurse, as shown in figure 14. Admittedly, under the standard interpretation, each such replacement would involve a dramatic increase in inefficiency, but the important point is that the resulting behaviour would in some sense still be correct.

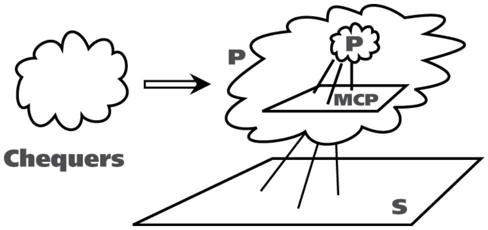


Figure 13 — Chequers via the MCP

5d Procedural Reflective Models

We are now in a position to unify the suggestion made at the end of section 5b, on having Q reflect upwards, with the insights embodied in the MCPs described in the previous section, to define what I will call the **procedural reflective model**. The fundamental insight arises from the eminent similarity between figures 10 and 11, on the one hand, compared with figures 13 and 14, on the other. These diagrams do not represent exactly the same situation, but the approach will be to converge on a unification of the two.

I said earlier that in order to satisfy the requirements on the Q of §5b we would need to provide a causally connected structural encoding of a procedural theory of our dialect (Lisp in this case) within the accessible structural field. In the immediately preceding section we have seen something that is *approximately* such an encoding: the metacircular processor. However—and here I refer back to the six properties of reflection set out in §2—in the normal course of events the MCP lacks the appropri-

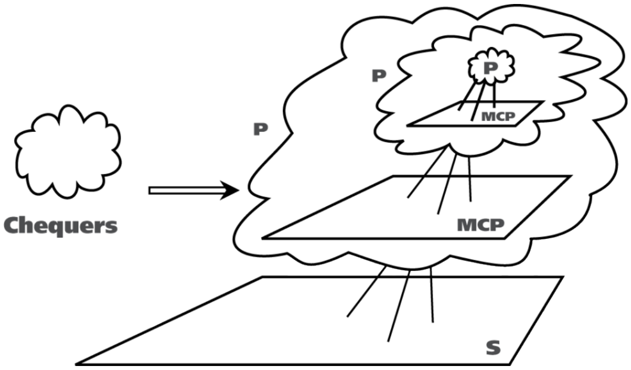


Figure 14 — Two layers of MCP

ate causal access to the state of P: whereas any possible state of Q *could* be procedurally encoded in terms of the metacircular process (i.e., given any account of the state of P we could retroactively construct appropriate arguments for the various procedures in the metacircular processor so that if that metacircular processor were run with those arguments it would mimic P in the given state), in the normal course of events the state of P will *not be so encoded*.

This similarity, however, does suggest the form of the solution.

Suppose that P were *never* run directly, but were *always* run in virtue of the explicit mediation of the metacircular processor—as, for example, in figure 13 and 14. Then at any point in the course of the computation, if that running of one level of the MCP were *interrupted,* and the arguments being passed around were used by some *other procedures,* they would be given just the needed information: causally connected and correct representations of the state of the process P prior to the point of reflection. The MCP would of course have to be modified in order to support such an interruption; the point however is that the MCP is already trafficking in the requisite causally connected representations.

There are however evident problems with this approach. First, if P were always run through the mediation of the metacircular processor MCP, P would as a result almost surely be unnecessarily inefficient. Second, as so far stated the proposal seems to deal with only one level of reflection. What if the code that was given these structural encodings of P's state was itself to reflect? This query suggests that providing a general mechanism for reflection would generate an infinite regress: not only should the MCP be used to run the base ("level 0") programs, but the MCP should be used to run the level 1 MCP. And so on. That is: *all* of an infinite number of MCPs should be run by yet further MCPs, ad infinitum.

Setting aside the obvious vicious regress for a moment, note that this seems otherwise to be a reasonable suggestion. The *potentially* infinite (i.e., indefinite) set of reflecting processes Q are almost indistinguishable in basic structure from the infinite tower of MCPs that would result. Furthermore the MCP' would contain just the correct structurally encoded descriptions of processor state. We would still need to modify the whole set of MCPs, so that an appropriate interruption or reflective act could make use of the tower of processes, but it is nevertheless evident that, to a first degree of approximation, this solution has the proper character.

The fundamental "trick" of 3-Lisp (i.e., of the model of procedural reflection being proposed) hinges on the fact that, it turns out, we can effectively *posit that the primitive reflective processor is engendered by an infinite number of recursive instances of the* MCP, *each running a version one level below*. That is: 3-Lisp will be defined to be isomorphic to that infinite limit. This turns out to be legitimate—i.e., the implied infinite regress is not after all problematic—since only a finite amount of information is encoded in it; at all but a finite number of the bottom levels, each MCP will merely be running a copy of the MCP. Because we, as the language designers, know exactly how the language runs, and because we also know what the MCP is like, we can provide this infinite numbers of levels, to use current jargon, *purely virtually*. As I will explain in detail in chapter 5, such a virtual simulation turns out to be perfectly well-defined.

Once the changes are made to support appropriate interruption and resumption at any arbitrary level, it becomes no longer appropriate to call the processor a *metacircular* processor, since it becomes inextricably woven into the fundamental architecture of the language (as will be explained in detail in chapter 5). This is why, as suggested above, I call it a *reflective processor*. Nonetheless its genealogical roots in the abstract idea of an infinite tower of metacircular processor should be clear.

To provide a little bit of concrete grounding for this suggestion, I will explain just briefly the "interruption adjustment" we will make in order to allow this architecture to be used.

3-Lisp supports what I will call **reflective procedures**—procedures that, when invoked, are run not at the level at which the invocation occurred, but one level higher in the **reflective hierarchy.** They are given, as arguments, *those structures that would have been passed around in the reflective processor, had it always been running explicitly*. The code for the resulting 3-Lisp reflective processor program is given in figure 15, in part so that it may be

```
(define READ-NORMALISE-PRINT
.. (lambda simple [level env stream]
..... (normalise (prompt&read level stream) env
........ (lambda simp1e [result]                              ; C-REPLY
........... (block (prompt&reply result level stream)
.................. (read-normalise-print level env stream))))))
(define NORMALISE
.. (lambda simple [struc env cont]
..... (cond [(normal struc) (cont struc)]
.......... [(atom struc) (cont (binding struc env))]
.......... [(rail struc) (normalise-rail struc env cont)]
.......... [(pair struc) (reduce (car struc) (cdr struc) env cont)]))
(define REDUCE
.. (lambda simple [proc args env cont]
..... (normalise proc env
........ (lambda simple [proc!]                               ; C-PROC!
........... (if (reflective proc!)
............... (↓(de-reflect proc!) args env cont)
............... (normalise args env
.................. (lambda simple [args!]                     ; C-ARGS!
..................... (if (primitive proc!)
........................ (cont ↑(↓proc! . ↓args!))
........................ (normalise (body proc!)
.................................. (bind (pattern proc!) args! (environment proc!))
.................................. cont))))))))
(define NORMALISE-RAIL
.. (lambda simple [rail env cont]
.... (if (empty rail)
........ (cont (rcons))
........ (normalise (1st rail) env
........... (lambda simple [first!]                           ; C-FIRST!
.............. (normalise-rail (rest rail) env
.................. (lambda simple [rest!]                     ; C-REST!
..................... (cont (prep first! rest!)))))))))
```

Figure 15 — The 3-Lisp Reflective Processor Program

compared with the (very similar) 2-Lisp meta-circular processor code given earlier in figure 12. The most important difference lies on a single line, underlined here for emphasis.

What is important about the underlined line is this: when a redex (application) is encountered whose CAR normalises to a reflective as opposed to standard procedure (the standard ones are called "simple"), the corresponding function, designated by the term ↓(de-reflect proc!), is run *at the level of the reflective processor,* rather than *by* the processor. In other words the inclusion of this single underlined line unleashes the full infinite reflective hierarchy.

Coping with that hierarchy will occupy part of chapter 5, where I explain this all in much more depth (including why the resulting virtual machine is in fact finite, and how it can be implemented). Just this much of an introduction, however, should convey, if only a glimpse of how reflection is possible, at least the architectural structure of a language that provides it.

5d Two Views of Reflection

The reader will have noted a tension between two ways in which I have characterised the form of reflection we are aiming at. On the one hand I have sometimes written as if there were a primitive and noticeable **reflective act**, which causes the processor to **shift levels** rather markedly (this is the explanation that best coheres with some of our pre-theoretic intuitions about reflective human thinking). On the other hand, I have also just written of an infinite number of levels of re1ective processors, each essentially implementing the one below—a story according to which it is not coherent either to ask at which level Q is running, or to ask how many reflective levels are running. On this "infinite tower" account, there is a strong some sense in which *all levels are running at once,* in exactly the same sense that both the Lisp processor inside your Lisp-based editor, and your editor itself, and the machine language code that underpins the implementation of Lisp, are all running at once, when you use the editor. It is of course not as if Lisp, the editor, and the machine language are running simultaneously in the sense of *side-by-side* or *independently*. This is not a parallel computing scheme being described. On the other hand, in each case one, being "interior" to the other, supplies the anima or agency of the outer one (machine language processor animating the Lisp processor, which in turn animates the editor). It is just this sense in which the higher levels in the 3-Lisp reflective hierarchy are always running: each of them is in some sense *within* (interior to) the processor at the level below it, in such a way that it thereby engenders its agency.

Call the account that views reflection as a case of a single locus of agency stepping between levels the **level-shifting** view. And call the other view that of an **infinite tower**. I will not take a principled view on which is correct; on the contrary, the architectural thesis behind 3-Lisp, and behind the model of reflection being proposed, can be understood as comprising two parts: (i) that they can be shown behaviourally equivalent, and thus (ii) that adopting the architecture of the tower view is an appropriate way to understand (and implement) the level-shifting view. For certain purposes one is simpler, for others the other.

Though perhaps more initially intuitive, the level-shifting account turns out to be more complex than the tower view. To illustrate it, consider the following account of what is involved in constructing a reflective dialect—in part by way of review, but also in order to suggest how it is that a practical reflective dialect could be finitely constructed.

1. As I have repeatedly said, in order to design a reflective language one must provide a complete theory of the given calculus expressed in its own language. I call this the reflective processor—it is required on both accounts.
2. You must arrange things so that, when the process reflects—i.e., when the locus of control shifts "upwards"—all of the structures used by the reflective processor (the formal structures designating the theoretical entities posited by the theory) are available for inspection and manipulation. In any particular case, these to-be-provided structures must correctly encode *the state that the processor was in prior to the reflective level-shift*, assuming that it had been running all the while (this is where the tower view provides structure and substance—fills in the technical details—for the level shifting view).
3. You must also ensure, when the (level-shifting) process comes to the point of "shifting down" again, that base-level processing is resumed *in accordance with the facts encoded in the structures being passed around at the immediately higher reflective level*.

As a minimal case, take a situation where the user process shifts upwards, but does nothing; and then shifts down again. At the point of shifting up, the situation should merely be one where the processor would process the reflective processor code explicitly, as if it had been doing so all along. At the point of shifting down, it would take up running the base-level code directly (i.e., non-reflectively), again *as if it had been doing that all along*, but also (of course it must be proved that these are equivalent) exactly in accord with the state of the structures being passed around in the reflective processor code at the point of down-shifting. Such a situation, in fact, is *so* simple that it could not be distinguished (except perhaps in terms of elapsed time) from pure non-reflective interpretation.

The situation would get more complex, however, as soon as the user is given any power. Two provisions in particular are crucial.

First, the whole purpose of a reflective dialect is to allow the user to have his or her own programs run along with, or in place of, or between the steps of, the reflective processor. One must in other words provide an abstract machine with the ability for the programmer to insert code—in convenient ways and at convenient times—at any level of the reflective hierarchy. Suppose, for example, we were to wish to have a particular λ-expression closed only in the dynamic environment of its use, rather than in the lexical environment of its definition (i.e., suppose we were to want "dynamic scoping" for a given λ-expression, even though lexical scoping is the system default). Needless to say, the reflective processor contains code that performs the requisite operations needed to implement the default behaviour for lexical closures. Given that programmer can assume that, upon reflection, the reflective processor code is being explicitly processed, he or she can supply, for the lambda expression in question, an appropriate alternate piece of code in the different actions are taken so as to provide it with dynamic scoping behaviour.. By simply inserting this code into the correct level, (s)he can use variables bound by the reflective model in order to fit gracefully into the overall processing regimen. Appropriate hooks and protocols for such insertion, of course, must be provided, but they need be provided only once. Furthermore, the reflective processor code (i.e., reflective model) will contain code showing how this hook is treated.

All of these requirements are met by the underlined line 18 in the reflective processor program of figure 15. That line indicates how the user code will be inserted, what context it will run it, what variables will be bound to what structures containing what information, etc.

Second, as well as providing for the arbitrary interpretation of special programs at the reflective level, the language designer must also enable the user to *modify* the explicitly available structures provided in the reflective model. Though this ability is easier to design than the former, its correct implementation is trickier. An example will make this clear. As already indicated, the 3-Lisp reflective processor deals explicitly with both environment and continuation structures. Upon reflecting, user programs can at will access these structures that, at the base level, are purely implicit. Suppose that a user writes reflective code that does two things. First, it modifies the environment structure being passed around at the first reflective level (e.g., suppose it changes the binding of a variable bound by some procedure that is running "somewhere up the stack," in the way that might be provided by a typically debugging package). Second, it changes the continuation structure (designating the continuation function) so as to cause some procedure that is currently running to, upon its return, bypass its immediate caller, and instead return its result to the procedure who called that procedure. Then, once it has effected these two changes, it "returns"—which is to say, it "drops back down" to other base-level code, and no longer runs at the reflective level.

I said above that, upon this kind of semantic or reflective descent, the base-level program will again be processed "directly." But of course it must be processed in such a way as to honour the changes indicated by these modified structures—not in the way that it would have been processed, prior to the reflection. The user's reflective modifications, in other words, must *matter*—must be *noticed*. This is the (downwards direction of) the *causal connection* aspect that is so crucial to true reflection.

53 General Comments

The details of the proposed architecture have emerged from detailed considerations of process reduction, computational semantics, and meta-circular processing. It is interesting to draw back and to see the extent to which the global properties of the resulting architecture match our pre-theoretic intuitions about reflection.

First, it is simple to see that the proposed architecture honours all six requirements laid out in section 2c:

1. It is causally connected and theory-relative;
2. It is theory-relative;
3. It involves an incremental "stepping back," rather than a full (and potentially vicious) instantaneous "reflexion";
4. Finer-grained control is provided over the processing of lower level structures;
5. It is only partially detached (3-Lisp reflective procedures are still in and part of 3-Lisp; they are still animated by the same fundamental agency, since if one level stops processing the reflective model, or some analogue of it, all the processors "below" it cease to exist): and
6. The reflective powers of 3-Lisp are primitively provided.

Thus in this sense at least it is fair to count the architecture a success.

Other questions—such as about the locus of self, the concern as to whether the potential to reflect requires that one always participate in the world indirectly rather than directly, and so forth—turn out to be about as difficult to answer for 3-Lisp as they are to answer in the case of human reflection. In particular, the solution I have proposed does not answer the question I posed earlier, about the identity of the reflected processor: is it P that reflects, or is it another process Q that reflects on P? The "reflected process" is neither quite the same process, nor quite a different process; it is in some ways as different as an *interior* process, except that since it shares the same structural field it is not as different as an implementing process. No more informative answer will be forthcoming until we define individuation criteria on processes much more precisely—and perhaps more strikingly, there seems no particular reason to answer the question one way or another. It is tempting (if dangerous) to speculate that the reason for these difficulties in the human case is exactly why they do not have answers in the case of 3-Lisp: they are not, in some sense, "real" questions. But it is premature to draw this kind of parallel; our present task is merely to clarify the structure of proposed solution.

6 Lisp as an Explanatory Vehicle

There are any number of reasons why it is important to work with a specific programming language, rather than abstractly and in general (for pedagogical accessibility, as a repository for emergent results, as an example to test proposed technical solutions, and so forth). Furthermore, commonsense considerations suggest that a familiar dialect, rather than a totally new formalism, would better suit our purposes. On the other hand there are no current languages that are categorically and semantically rationalised in the way that the proposed theory of reflection demands; according to the "reflection is intelligibly implementable only on a semantically clarified basis" mandate, it is not an option to endow any extant system with reflective capabilities without first subjecting it to substantial modification. It would be possible to present some system embodying all the necessary modifications and features, but it would be difficult for the reader to sort out which architectural features were due to what concern. In this dissertation, therefore, I have adopted the strategy of presenting a reflective calculus in two steps: first, by modifying an existing language to conform to the outlined semantical mandates; and second, by extending the resulting rationalised language with reflective capabilities.

Once this overall plan has been agreed, the question arises as to what language should be used as a basis for this two-stage development Since my present concern is with *procedural* rather than with *general* reflection, the relevant class of potential languages includes essentially all programming languages, but excludes exemplars of the declarative tradition: logic, the λ-calculus, specification and representation languages, and so forth.[x] Furthermore, we

[x] In the original dissertation, the following parenthetical comment was inserted at this point: "It is important to recognise that the suggestion of constructing a reflective variant of the λ-calculus represents a category error." Especially given the first half of the sentence, it is hard to know what

need a programming language—a procedural calculus—with at least the following properties:

1. Though not a formal requirement, it helps for the chosen language to be *simple*. By itself reflection is complicated enough that, especially as an initial illustration of the coherence and power of the architecture, it seems recommended to introduce it into a formalism of minimal internal complexity;
2. It must be possible to access program structures as first-class elements of the language's structural field;
3. Meta-structural primitives must be provided (the ability to *mention* structural field elements, such as data structures and variables, as well as to *use* them); and
4. The underlying architecture should facilitate the embedding, within the calculus, of the procedural components of its own meta-theory.

The second property could be added to a language: we could devise a variant on ALGOL, for example, in which ALGOL programs were made an extended data type, but Lisp already possesses this feature. In addition, since (in the formal semantical analysis presented in following chapters) I will use an extended λ-calculus as the meta-language, it is natural to use a procedural calculus that is functionally oriented. Finally, although full-scale modern Lisps are as complex as any other languages, both Lisp 1.6 and Scheme have the requisite simplicity.

Lisp has other recommendations as well. Because of its support of accessible program structures, it provides considerable evidence of exactly the sort of inchoate reflective behaviour that it has been my aim to reconstruct The explicit use of EVAL and APPLY, for example, provides considerable fodder for subsequent discussion, both in terms of what they do well and how they are confused. In chapter 2, for example, I describe half a dozen types

exactly this meant (if anything true); and in point of fact I informally defined a reflective version of the λ-calculus a couple of years later, as a vehicle in terms of which to explain reflection to my colleague Jon Barwise. I have therefore omitted it from this version.

of situation in which a standard Lisp programmer would be tempted to use these meta-structural primitives, only two of which in the deepest sense have anything to do with the explicit manipulation of expressions; the other four, I will argue, ought to be treated directly in the object language—and their use of metastructural machinery understood to be no more than a "work-around" for fundamental failures in Lisp's original design.[x] And finally, and non-trivially, Lisp is the *lingua franca* of the AI community; this fact alone makes it an eminent candidate.

6a 1-Lisp as a Distillation of Current Practice

The decision to use Lisp as a base does not solve all of cur problems, since the name "Lisp" still refers to a wide range of languages and dialects. For purposes of this dissertation it has seemed simplest to define a simple kernel, not unlike Lisp 1.6, as a basis for further development, in part to have a fixed and well-defined target to set up and criticise, and in part so that I can collect into one dialect the features that prove most important for subsequent analysis. I take Lisp 1.6 as the primary source for the result, which I have called 1-Lisp, although some facilities I will ultimately want to examine as (often inchoate) examples of reflective behaviour—such as CATCH and THROW and QUIT—have been included, along with the repertoire of behaviours manifested in McCarthy's original design. Similarly, I have included macros as a primitive procedure type, as well as intensional and extensional procedures of the standard variety ("call-by-value" and "call-by-

[x] In a colloquium in the Artificial Intelligence Laboratory at SRI International, in the spring of 1982, I have one of the very first talks on 3-Lisp. As it happened, John McCarthy (inventor of Lisp, and designer of Lisp 1.6) attended. Though as a young student I was nervous about making this claim in front of him, I nevertheless proceeded with what I had planned to say, and claimed that, according to my analysis, traditional Lisp's dynamic scoping protocols were a "mistake," to which quotation and other metastructural manoeuvrings were a partial work-around—in particular providing a way of handing closures "downwards," though there was no way to pass them "upwards" (in terms of the usual notion of a control stack; this has nothing to do with the reflective hierarchy).

To my surprise and considerable relief, John McCarthy very graciously agreed.

name," in standard computer science parlance, although I avoid these terms, since I reject the notion of "value" entirely).

It turns out not to be entirely simple to present 1-Lisp. given my theoretical biases, since so much of what I will ultimately reject about it comes so quickly to the surface in explaining it. However I have felt that it is important to present this formalism without modification, because of the role I ask it to play in the structure of the overall argument. In particular, my desideratum for the dialect is not that it be clean or coherent, but rather that it serve as a vehicle in which to examine a body of practice suitable for subsequent reconstruction. To the extent that I make empirical claims about semantic reconstruction, I use 1-Lisp as evidence in its role as being a model of all extant Lisp practice. It is theoretically critical, given this role, that I leave this practice as intact as possible, free of my own theoretical biases. Even though it is a dialect of my own design, therefore, I have intentionally but uncritically forged it in terms of received notions of evaluation, lists, free and global variables, and so forth.

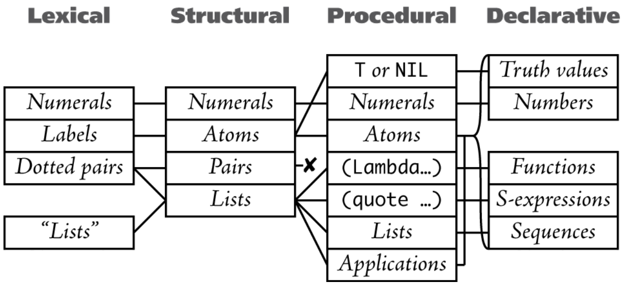


Figure 16 — 1-Lisp Category Structure

As an example of the style of analysis to be engage in, figure 16 gives a diagram of the 1-Lisp category structure—to be contrasted with the category structure of 2-Lisp and 3-Lisp, which has been designed to satisfy the category alignment mandate. The intent of the diagram is to show that in 1-Lisp (as in any computational calculus) there are a variety of ways in which structures or s-expressions may be categorised—represented in turn by each of the vertical columns. The point I am attempting to demonstrate is the (unnecessary) complexity of interaction between these various categorical decompositions.

Consider each of these various 1-Lisp categories in brief. The first column *(notational)* is categorized by the lexical categories accepted by the reader (including strings that are parsed into notations for numerals, lexical atoms, and "list" and "dotted-pair" notations for pairs). Another categorization *(structural)* is in terms of the primitive types of s-expression (numerals, atoms, and pairs); this is the categorisation typically revealed by the primitive structure typing predicates (in 1-Lisp I call this procedure TYPE, but it is traditionally encoded in an amalgam of ATOM and NUMBERP). A third traditional categorisation *(derived structure)* includes not only the primitive s-expression types but also the derived notion of a *list*—a category built up from some pairs (those whose CARS are, recursively, lists) and the atom NIL. A fourth taxonomy (labeled *procedural consequence)* is embodied by the primitive processor: thus 1-Lisp 's evaluation processor (EVAL) sorts structures into various categories, each handled differently. This is the "dispatch" categorization that one typically finds at the top of metacircular definitions of EVAL and APPLY. In most Lisp metacircular processors six categories are discriminated:

1. The self-evaluating atoms T and NIL;
2. The numerals;
3. The other atoms, used as variables or global function designators, depending on context;
4. Lists whose first clement is the atom LAMBDA, used to encode applicable functions;
5. Lists whose first clement is the atom QUOTE; and
6. Other lists, which in evaluable positions represent function application.

Finally, the fifth taxonomy *(declarative import)* has to do with declarative semantics—i.e., discriminates categories of structure based on their *signifying* different sorts of semantic entities. Once again a different category structure emerges: applications and variables can signify semantic entities of arbitrary type *except that they cannot designate procedures* (since 1-Lisp is first-order); the atoms T and NIL signify Truth and Falsity; general lists, plus again (in different contexts) the atom NIL, signify enumerations (sequences): the numerals signify numbers; and so on and so forth.

The reason why the demerits of this non-alignment of categories multiply in a reflective dialect is that reflective programs need to know about all of them, in different situations and for different purposes—and also about the relationships between and among them (as, impressively, all human Lisp programmers do). And remember, too, that as one climbs from reflective level 1 to yet higher reflective levels, the combinatorics of non-alignment would multiply correspondingly. I need not dwell on the evident disarray that would likely result.

One other example of 1-Lisp behaviour will be illustrative. I have mentioned above that 1-Lisp requires the explicit use of APPLY in a variety of circumstances. These include the following:

1. When an argument expression designates a function *name*, rather than a function—as for example in

   ```
   (APPLY (CAR '(+ - *)) '(2 3))
   ```

2. When the arguments to a multiple-argument procedure are designated by a single term, rather than designated individually. Thus if X evaluates to the list (3 4), one must use (APPLY '+ X) rather than (+ X) or (+ . X).
3. When a function is designated by a variable rather than by a global constant. Thus one must use:

   ```
   (LET ((FUN '+)) (APPLY FUN '(1 2)))
   ```

   rather than the simpler:

   ```
   (LET ((FUN '+)) (FUN 1 2))
   ```

4. When the arguments to a function are "already evaluated", since APPLY, although itself extensional (it is an "EXPR"), does not re-evaluate the arguments even if the procedure being applied is an EXPR. Thus one uses:

   ```
   (APPLY '+ (LIST X Y))
   ```

   rather than:

   ```
   (EVAL (CONS '+ (LIST X Y)))
   ```

As I will show, in 2-Lisp and 3-Lisp only the first of these will require explicitly mentioning the processor function by name, because it inherently deals with the *designation of expressions*, rather than with the designation of their referents. *Because of their cate-*

gory alignment, 2-Lisp and 3-Lisp treat the other three cases adequately in the object language.

6b The Design of 2-Lisp

Though it meets the criterion of simplicity, 1-Lisp provides more than ample material for further development, as the previous examples suggest. Once I have introduced it, as mentioned earlier, I subject it to a semantical analysis that leads us into an examination of computational, semantics in general, as described in the previous section. The search for semantical rationalisation, and the exposition of the 2-Lisp that results, occupies a substantial part of the dissertation, even though the resulting calculus still fail to meet the requirements of procedural reflection (as befitting the underlying thesis that reflection is relatively straightforward, once these semantical issues are taken care of). I discussed what semantic rationalisation comes to in general in a previous section (§■■); here I sketch how its mandates are embodied in the design of 2-Lisp.

The most striking difference between 1-Lisp and 2-Lisp is that the latter rejects evaluation in favour of independent notions of *simplification* and *reference.* Thus, 2-Lisp 's processor is not called `EVAL`, but `NORMALISE`, where by *normalisation I* refer to a particular form of expression simplification that takes each structure into what I call a *normal-form* designator of that expression's referent (making normalization designation-preserving). Details are provided in chapter 4, but a sense of the resulting architecture can be given here.[x]

Simple object level computations in 2-Lisp (those that do not involve meta-structural structures designating other elements of the Lisp field) are treated in a manner that looks very similar to 1-Lisp. The expression `(+ 2 3)`, for example, normalises to `6`, and the expression `(= 2 3)` to `$F` (the primitive 2-Lisp boolean constant designating falsity). On the other hand an obvious superfi-

[x] Somewhere I should talk about processing "honouring" such semantics; which is explicit in logic; only implicit (part of practice) in CS; but made explicit again here. Cf. Mike's Amala proposals, too; it should be highlighted in the overall introductory annotation (which I haven't written yet).

cial difference is that in 2-Lisp *meta-structural* terms are not automatically dereferenced. Thus the quoted term `'X`, which in 1-Lisp would evaluate to `X`, normalises in 2-Lisp to itself. Similarly, whereas `(CAR '(A . B))` would evaluate in 1-Lisp to `A`, in 2-Lisp it normalises to `'A`. Similarly, in 1-Lisp `(CONS ‘A ‘B)` evaluates to the pair `(A . B)`; in 2-Lisp the corresponding expression would yield the handle `'(A . B)`.

From these almost trivial examples, one might be tempted to embrace the following idea: that the 2-Lisp processor is just like the 1-Lisp processor, except that it puts a quote back on before returning the result. But that is ill-advised; the difference, more theoretically motivated, is more substantial in terms of structure, procedural protocols, and semantics. For starters 2-Lisp, like Scheme, is statically-scoped and higher-order; function-designating expressions may be passed as regular arguments. 2-Lisp is also structurally different from I-Lisp; there is no derived notion of *list*, but rather a primitive data structure called a **rail** that serves the function of designating a sequence of entities (pairs are still used to encode function applications). What in 1-Lisp are called "quoted expressions" correspond to the primitive structural type **handle**, not to applications framed in terms of a (pseudo) `QUOTE` procedure; they are also canonical (one per structure designated). The 2-Lisp notation `'X`, in particular, is not an abbreviation for `(QUOTE X)`, but rather the primitive notation for the handle that is the unique normal-form designator of the atom `X`. There are other notational differences as well: rails are expressed with square brackets (thus the expression ‘`[1 2 3]`’ notates a rail of three numerals that in turn designates a sequence of three numbers), and expressions of the form

$$\texttt{(F } A_1\ A_2\ \ldots\ A_k\texttt{)}$$

expand not into

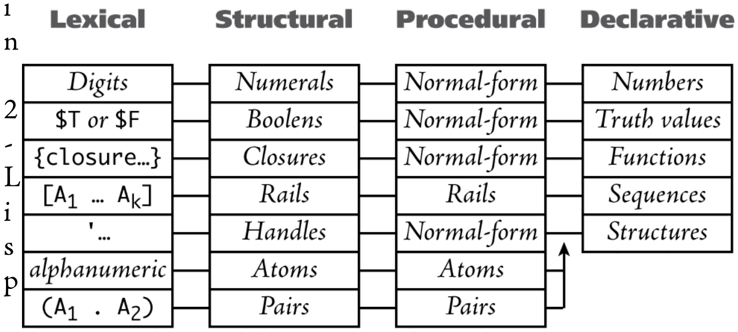
$$\texttt{(F . (}A_1\texttt{ . (}A_2\texttt{ . (}\ldots\texttt{ . (}A_k\texttt{ . NIL)}\ldots\texttt{))))}$$

but instead into

$$\texttt{(F . [}A_1\ A_2\ \ldots\ A_k\texttt{])}$$

The category structure of 2-Lisp is summarized in figure 17.

Closures, which have historically been treated as rather curious entities somewhere in between functions and expressions, emerge in 2-Lisp as



Lexical	Structural	Procedural	Declarative
Digits	*Numerals*	*Normal-form*	*Numbers*
$T *or* $F	*Boolens*	*Normal-form*	*Truth values*
{closure…}	*Closures*	*Normal-form*	*Functions*
[A_1 … A_k]	*Rails*	*Rails*	*Sequences*
'…	*Handles*	*Normal-form*	*Structures*
alphanumeric	*Atoms*	*Atoms*	
(A_1 . A_2)	*Pairs*	*Pairs*	

Figure 17 — 2-Lisp Category Structure

standard expressions; in fact I *define* the term '**closure**' to refer to a *normal-form function designator*. Not only are closures pairs, but all normal-form pairs are closures, illustrating once again the category alignment that permeates the design.

As stated above, all 2-Lisp normal-form designators are not only *stable* (self-normalising), but also *side-effect free* and *context-independent*. A variety of facts emerge from this result. First, the primitive processor procedure (NORMALISE) can be proved to be *idempotent* in terms of both result and total effect:

∀S [(NORMALISE S) = (NORMALISE (NORMALISE S))] (10)

Consequently, as in the λ-calculus, the result of normalising a constituent (in an extensional context) in a composite expression may be substituted back into the original expression, in place of the non-normalized expression, yielding a *partially simplified* expression having the same designation and same normal-form as the original. So support for "partial evaluation" is in some sense an automatic feature of the two dialects. In addition, in code-generating code such as macros and debuggers and so forth, there is no need to worry about whether an expression has *already* been processed, since second and subsequent processings will never cause any harm (nor, as it happens, will they take any time).

All of the foregoing facts can in some sense be considered to be *simplifications* embedded in the design of 2-Lisp. Most of 2-Lisp 's complexities emerge only when one consider forms that designate other semantically significant forms. The intricacies of such "level-crossing" expressions are the stock-and-trade of a reflective system designer, and only by setting such issues straight *before* we consider reflection proper will we face the latter task adequately prepared.

Primitive procedures called NAME and REFERENT (notationally abbreviated '↑' and '↓') are provided to mediate between sign and significant (they must be primitive because without them the processor provably remains semantically flat); thus (taking '⇒' to mean "normalises to"):

```
↑3   ≡ (NAME 3)      ⇒ '3
↓'A  ≡ (REFERENT 'A) ⇒ 'A
```

The issue of the explicit use of APPLY, mentioned in the discussion of 1-Lisp, above, is instructive to examine in the 2-Lisp context, since it manifests both the structural and the semantic differences between 2-Lisp and its precursor dialect. In 1-Lisp, the functions EVAL and APPLY mesh in a well-known mutually-recursive fashion. Evaluation is uncritically thought to be defined over *expressions*, but it is much less clear what application is defined over. On one view, APPLY is a functional that maps functions and (sequences of) arguments onto the value of the function at that argument position—thus making it a second (or higher) order function. On another view, APPLY takes two *expressions* as arguments, and has as its value a third expression that *designates* the value of the function designated by the first argument at the argument position designated by the second. In 2-Lisp I will call the first of these notions **application** and the second **reduction** (the latter in part because the word suggests an

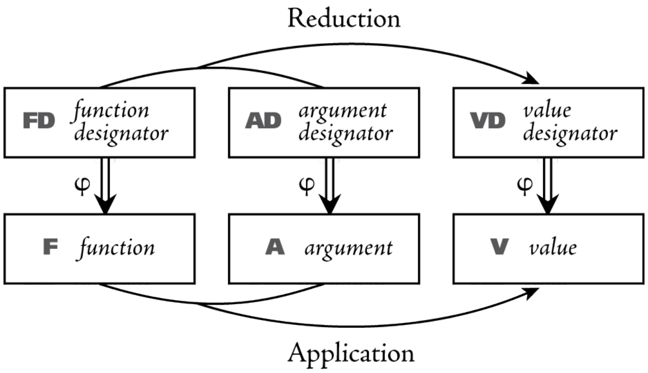


Figure 18 — Reduction and Function Application

operation over expressions, and in part by analogy with the β-reduction of Church.[22] Current Lisp systems are less than lucid regarding this distinction (in MacLISP , for example, the *function* argument is an expression, whereas the *arguments* argument is not an expression, nor is the value). The position I will adopt is depicted in figure 18 (to be explained more fully in chapter 3):

The procedure REDUCE, together with NORMALISE will of course play a major role in the characterisation of 2-Lisp, and in the subsequent construction of the reflective 3-Lisp. It is worth noting, however, that although it would be trivial to do so, there is no reason to define a designator of the APPLY function, since any term of the form:

```
(APPLY FUN ARGS)
```

would be equivalent in both designation and effect (i.e., would be equivalent in full computational significance) to:

```
(FUN . ARGS)
```

In contrast, since it is a meta-structural function, REDUCE is neither trivial to define (as is APPLY) nor recursively empty.

A summary of the most salient differences between 2-Lisp and 1-Lisp is provided in the following list:

1. **Scoping:** 2-Lisp is lexically scoped, in the sense that variables free in the body of a LAMBDA form take on the bindings in force in their statically enclosing context, rather than from the dynamically enclosing context at the time of function application.
2. **Functions:** Functions are first-class semantical objects, and may be designated by standard variables and arguments. As a consequence, the function position in an application (the CAR of a pair) is both procedurally and declaratively "extensional," and thus normalised in exactly the same way as argument positions.
3. **Simplification:** Evaluation is rejected in favour of independent notions of *simplification* and *reference*. The primi-

[22] Church (1941).

tive processor is a particular kind of *simplifier.* rather than being an *evaluator.* In particular, it *normalises* expressions, returning for each input expression a normal-form co-designator.

4. **Declarative Semantics:** A complete theory of declarative semantics is postulated for all s-expressions. prior to and independent of the specification of how they are treated by the processor function—a pre-requisite to the claim that the processor is designation-preserving):.
5. **Closures:** Closures—normal-form function designators—are valid and inspectable s-expressions.
6. **Normal Form:** Though not all normal-form expressions are canonical (functions, in particular, may have arbitrarily many distinct normal-form designators), nevertheless they are all *stable* (self-normalising), *side-effect free,* and both declaratively and procedurally *context independent.*
7. **Semantically Flat:** The primitive processor (designated by `NORMALISE`) is *semantically flat;* in order to shift level of designation one of the explicit semantical primitives `NAME (↑)` or `REFERENT (↓)` must be applied.
8. **Category Alignment:** 2-Lisp is *category-aligned* (as indicated in figure 17, above): there are two distinct structural types, *pairs* and *rails,* that respectively encode function applications and sequence enumerations. There is in addition a special two-element structural class of boolean constants. There is no distinguished atom `NIL`.
9. **Binding:** Variable binding is *co-designative,* rather than being either *evaluative* or *designative,* in the sense that a variable *normalises* to what it is *bound to,* and therefore designates the referent of the expression to which it is bound. Although I will speak of the *binding* of a variable, and of the *referent* of a variable, I will not speak of a variable's *value,* since that term conflates these two notions.
10. **Identity:** Identity considerations on normal-form designators are as follows: the normal-form designators of truth-values, numbers, and s-expressions (the booleans, numerals, and handles, respectively) are unique. Normal-form

designators of sequences (rails) and functions (pairs) are not. No atoms are normal-form designators of anything; therefore the question does not arise in their case.

11. **LAMBDA:** The use of LAMBDA is purely an issue of abstraction and naming, and is completely divorced from procedural *type* (extensional, intensional, macro, and so forth).

As soon I we have settled on the definition of 2-Lisp, however, I will begin to criticise it. In particular, I will provide an analysis of how 2-Lisp fails to be appropriately reflective, in spite of its semantical cleanliness.

A number of problems in particular emerge as troublesome. First, it will turn out that the clean *semantical* separation between meta-levels is not yet matched with a clean *procedural* separation. For example, too strong a separation between environments, with the result that intensional procedures become extremely difficult tn use, shows that in one respect, 2-Lisp 's inchoate reflective facilities suffer from insufficient causal connection. On the other hand, awkward interactions between the control stacks of inter-level programs will show how, in other respects, there is *too much* connection. In addition, although I will demonstrate a metacircular implementation of 2-Lisp in 2-Lisp, and will provide 2-Lisp with explicit names for its basic interpreter functions (NORMALISE and REDUCE), these two facilities will remain utterly unconnected—an instance of a general problem to be discussed in chapter 3 on reflection in general.

6c The Procedurally Reflective 3-Lisp

From this last analysis will emerge the design of 3-Lisp, a procedurally reflective Lisp and the last of the dialects to be considered here.

As presented in chapter 5, 3-Lisp differs from 2-Lisp in a variety of ways.

1. The fundamental *reflective act* is identified and accorded tbe centrality it deserves in the underlying definition.
2. Each reflective level is granted its own environment and continuation structure, with the environments and continuations of the levels below it accessible as first-class ob-

jects (inheriting a Quinean stamp of ontological approval, since they can be the values of bound variables).

3. As mentioned in the earlier discussion these environments and continuations are theory relative. The (procedural) theory is embodied in the 3-Lisp reflective model, a causally connected variant on the metacircular interpreter of 2-Lisp discussed in section 3.
4. Surprisingly, the integration of reflective power into the metacircular—now reflective—model is itself extremely simple (though to *implement* the resulting machine is not trivial).
5. Reflecting its more complete nature, in a number of ways 3-Lisp is notably simpler than 2-Lisp.

Once all these moves have been taken it will be possible to Inerge the explicit reflective version of `NORMALISE` and `REDUCE`, and the similarly named primitive functions. In other words the 3-Lisp reflective model unifies what in 2-Lisp were separate: primitive names for the underlying processor, and explicit metacircular programs demonstrating the procedural structure of that processor.

It was a consequence of defining 2-Lisp in terms of `NORMALISE`, a species of simplification, that the 2-Lisp processor is "semantically flat": the semantical level of an input expression is always the same as that of the expression to which it simplifies.. An even stronger claim holds for function application. Except in the case of the explicit level-shifting functions `NAME` (↑) and `REFERENT` (↓), the semantical level of the result is also the same as that of all of the arguments. This is all evidence of the effort to drive a wedge between simplification and *de-referencing* mentioned earlier. 3-Lisp inherits this semantical characterisation; note that it remains true *even in the case of reflective functions*.

A semantically-flat (fixed-level) processor of this form—one of the reasons 2-Lisp was designed this way—enables an important move: it becomes possible, though only in an approximate sense, to identify *declarative meta levels* with *procedural reflective levels*. This does not quite have the status of a *claim*, because it is virtually mandated by the Knowledge Representation Hypothesis (furthermore, the correspondence is somewhat asymmetric: declarative levels can be crossed within a given reflective level, but reflective shifts always involve shifts of designation). But it is instructive to realise that we have been able to identify the reflective act (that makes available the structures encoding the processing state and so forth) with two shifts: (i) the shift from objects to their names, and (ii) the shift from tacit aspects of the background to objects. *Reification*, that is, *emerges as the first form of semantic ascent*. Thus what was *used* prior to reflection is *mentioned* upon reflecting; what was *tacit* prior to reflection becomes *used* upon reflection.[x] When this behaviour is combined with the ability for reflection to recurse, we are able to lift structures that are normally tacit into explicit view in one simple reflective step; we can then obtain access to designators of those structures in another.

Later in the dissertation both the 3-Lisp reflective model, and a MacLISP implementation of it, will be provided by way of definition. In addition, some hints will be presented of the style of semantical equation that would be required for a traditional denotational-semantics style account of 3-Lisp —though it is important to admit that a full semantical treatment of procedural reflection in general or of 3-Lisp in particular has yet to be worked out.

In a more pragmatic vein, however, and in part to show how 3-Lisp satisfies many of the *desiderata* that motivated the original definition of the concept of reflection, I will present a number of

[x] Although I did not pay a great deal of attention to this claim at the time the dissertation was written, I was very struck by it when I came to realize it. It not only influenced the approach to real-world ontology that is sketched in O3, but it also infected the ideas I was mulling on, at the time, about fusing higher-order and intensional "objectification" levels in Mantiq.

I still believe that a substantial issue remains lurking here, with which a proper theory of cognition should come to grips: relations between and among processes of (i) reification—leading us to find the world intelligible in terms of objects; (ii) semantic ascent—generating quotation, meta-level concepts and expressions, and other forms of symbolic or cognitive "mention"); and (iii) the use of higher-order structures (such as higher-order functions). In our formal efforts to be rigorously clear about the *differences* among these notions, we sometimes fail to recognize their *similarity*—and more seriously, what may be their common genealogy.

examples of programs defined in 3-Lisp: a variety of standard functions that make use of calls to the processor, access to the implementation (debuggers, "single-steppers," and so forth), and non-standard "evaluation" (processing) protocols. The suggestion will be made that the case with which these powers can be embedded in "pure" programs recommends 3-Lisp as a plausible dialect in its own right. Nor is this simply a matter of using 3-Lisp as a theoretical vehicle in which to model or implement these various constructs, or of showing that such models fit naturally and simply into the 3-Lisp dialect (as a simple continuation-passing scheme can for example be shown to be adopted in Scheme). The claim is stronger: that such functionality can be naturally embedded in 3-Lisp in a manner that allows it to be congenially mixed (without pre-processing or pre-compilation) with simpler, more standard forms of practice. Without the user normally having to use (or even understand) explicit continuation-passing style, nonetheless, at any point in the course of the computation, the applicable continuation is easily and explicitly available (upon reflection) for any programs that wish to deal with such things directly. Similar remarks hold for other aspects of the control structure and environment

One final comment about the 3-Lisp architecture will relate it to the two views on reflection—"level-shifting" and "infinite-tower"—mentioned at the end of section 5. Modulo the amount of time it takes, processing mediated by the 3-Lisp reflective model is guaranteed to yield indistinguishable behaviour (at least from a non-reflective point. of view—there are subtleties here) from basic, non-reflected processing. It is this fact that allows us to make the abstract claim that 3-Lisp runs in virtue of an infinite number of levels of reflective models all running at once. by an (infinitely fleet) overseeing processor running at level ∞. The resulting infinite abstract machine is well defined, for it is of course behaviourally indistinguishable from the perfectly finite 3-Lisp that will already have been laid out (and implemented). For some purposes 3-Lisp is most easily described in terms of this infinite tower—and in some ways, too, it is the easiest model for the 3-Lisp programmer to have in mind, when writing programs.. Such a programmer can write programs to be interpreted at any reflective level, and cannot tell that all infinitude of levels are not being run (the implementation surreptitiously constructs them and places them in view each time the user's program steps back to view them), such a characterisation is usually more illuminating than talk of the processor "switching back and forth from one level to another". In terms of mathematical analysis, treating 3-Lisp as a purely formal object, the infinite tower characterisation would also be more likely to be preferred. On the other hand, when taken as a model of psychologically intuitive reflection—based on a vague desire to locate the *self* of the machine at some level or other—the language of level-shifting seems more highly recommended. Level-shifting is also a major and constant concern for anyone person who designs and constructs a 3-Lisp implementation.

6d Reconstruction Rather Than Design

2-Lisp and 3-Lisp can claim to be dialects of Lisp only on a generous interpretation. Both dialects are unarguably more different from the original Lisp 1.6 than are all other dialects that have previously been proposed, including for example Scheme, MDL, NIL, SEUS, MacLISP, InterLISP, and Common Lisp.[23]

In spite of this difference, however, I view it as important to the exercise to call these languages Lisp. The aim in developing them has not been simply to propose some new variants in a grand tradition, perhaps better suited for a certain class of problem than others that have gone before. Rather—and this is one of the reasons that this dissertation is as long as it is—it is my claim that the architecture of these new dialects, in spite of its difference from that of standard Lisps, *is a more accurate reconstruction than has heretofore been provided of the underlying coherence that organises our communal understanding of what Lisp is.* I am making an empirical claim, in other words—a claim that should ultimately be judged as right or wrong. Whether 2-Lisp or 3-Lisp are

[23] Scheme is reported in Sussman and Steele (1975) and in Steele and Sussman (1978a); MDL in Galley and Pfister (1975), NIL in White (1979), MacLISP in Moon (1974) and Weinreb & Moon (1981), and InterLISP in Teitelman (1978). Common Lisp and SEUS are both under development, as this is being written, and have not yet been reported in print, so far as i know (personal communication with Guy Steele and Richard Weyhrauch).

better than previous Lisps is of course a matter of interest on its own, but it is not the thesis that this dissertation has set out to argue.

6 Remarks

6a Comparison with Other Work

Although I know of no previous attempts to construct eitller a semantically rationalised or a reflective computational calculus, the research presented here is of course dependent on, and related to, a large body of prior work. There are in particular four general areas of study with which this project is best compared:

1. Investigations into the meta-cognitive and intensional aspects of problem solving (this includes much current research in Artificial Intelligence);
2. The design of logical and procedural languages (including virtually all of programming language research, as well as the study of logics and other declarative calculi);
3. General studies of semantics (including both natural language and logical theories of semantics, and semantical studies of programming languages); and
4. Studies of self-reference, of the sort that have characterised much of metamathematics and the theory of computability throughout this century, particularly since Russell, and including the formal study of the paradoxes, the Gödel incompleteness results, and so forth.

I will make detailed comments about connections between this project and such other work throughout the discussion (for example in chapter 5 I will compare the reflective sense of "self-reference" with the notion traditionally studied in logic and mathematics), but some general comments can be made here.

Consider first the meta-cognitive aspects of problem-solving, of which the dependency-directed deduction protocols presented by Stallman and Sussman, Doyle, McAllester, and others are an illustrative example.[24] This work depends on explicit encodings, in some form of meta-language, of information about object-level structures, used to guide a deduction process. Similarly, the meta-level rules of Davis in his TEIRESIUS system,[25] and the use of meta-levels rules as an aid in planning,[26] can be viewed as examples of inchoate reflective problem solvers. Some of these expressions are primarily procedural in intent,[27] although declarative statements (for example about dependencies) are perhaps more common, with respect to which particular procedural protocols are defined.

The relationship of the current project to this type of work is more one of support than of direct contribution. I do not present (or even hint at) problem solving strategies involving reflective manipulation, although the fact that others are working in this area has certainly been a motivation for my research. Rather, I attempt to provide a rigorous account of the particular issues that have to do simply with providing *facilities for reflection*, independent of *what such facilities are then used for*. An analogy might be drawn to the development of the λ-calculus, recursive equations, and Lisp, in relationship to the use of these formalisms in mathematics, symbolic computation, and so forth: the former projects provide a language and architecture, to be used reliably and perhaps without much conscious thought, as the basis for a wide variety of applications. The present dissertation will be successful not if it forces everyone working in meta-cognitive areas to think about the architecture of reflective formalisms, but almost the opposite: if it allows them to forget that the technical details of reflection were ever considered to be problematic. Church's α-reduction was a successful manoeuvre precisely because it means that one can treat the λ-calculus in the natural way; I hope that my treatment of reflective procedures will enable those who use 3-Lisp or any subsequent reflective dialect to treat "backing-off' in what they take to be "the natural way."

The "reflective problem-solver" reported by Doyle[28] deserves a

[24] Stallman and Sussman (1977), de Kleer et al. (1977).

[25] Davis (1980)

[26] Stefik (1981a and 1981b).

[27] de Kleer et al. (1977).

[28] Doyle (1981).

special comment. Again, I provide an underlying architecture which might facilitate his project, without actually contributing solutions to any of his particular problems about how reflection should be effectively used, or when its deployment is appropriate. Doyle’s envisaged machine is a full-scale problem solver; it is also (so at least he argues) presumed to be large, to embody complex theories of the world, and so forth. In contrast, 3-Lisp is not a problem solver at all (all the user is “given” is a language—very much in need of programming); it embodies only a small procedural theory of itself, and it is really quite small. As well as these differences in goals there are differences in content (I for example endorse a set of reflective levels, rather than any kind of true instantaneous self-referential “reflexive” reasoning); it is difficult, however, to determine with very much detail what his proposal comes to, since his report is more suggestive than final.

Given that 3-Lisp is not a problem solver of the sort Doyle proposes, it is natural to ask whether it would be a suitable language in which Doyle might *implement* his system. There are two different kinds of answer to this question, depending on how he takes his project.

If, on the one hand, Doyle is proposing a design of a complete computational architecture (i.e., a process reduced in terms of an ingredient processor and a structural field), and wishes to *implement* it in some convenient underlying language, then 3-Lisp ’s reflective powers will not in themselves immediately engender corresponding reflective powers in the virtual machine that he implements. Reflection, as I have been at considerable pains to demonstrate, is first and foremost a semantical phenomenon, and *semantical properties*—designation and normalisation protocols and reflection and the rest—*do not cross implementation boundaries* (this is one of the great powers, but also a very serious limitation, of implementation).[x] 3-Lisp would be useful in such a project to

[x] This is a very serious issue. Suppose that architecture or virtual machine Y is implemented on top of language or system X. The question has to do with which of various properties P_i exemplified by X (the underlying system) are “inherited” by—i.e., true of—system Y, in virtue of the implementation relation holding between them. The answers are complex, and illuminating. There is no way that Y can be a “real-time” system, for example (in the sense of providing metric guarantees about certain kinds of behaviour,

the extent that it is generally a useful and powerful language, but it is important to recognise that its reflective powers cannot be used directly to provide reflective capabilities in other architectures implemented on top of it.

There is an alternative strategy open to Doyle, however, by which he could use 3-Lisp 's reflective powers more directly. If, rather than defending a generic reflective architecture, he more simply intended to show how a particular kind of reflective reasoning was useful, he could perhaps construct such behaviour in 3-Lisp, and thus use its reflective capabilities rather directly. There are consequences of this approach, however: he would have to accept 3-Lisp structures *and semantics*, including among other things the fact that it is purely a procedural formalism. It would not be possible, in other words, to encode a full descriptive language on top of 3-Lisp, and then use 3-Lisp 's reflective powers to reflect in the general sense with these descriptive structures. If one aims to construct a general or purely descriptive formalism, one would have to make that architecture reflective on its own.

None of these conclusions stand as criticisms of 3-Lisp. They are entailed by fundamental facts about computation and semantics—not limitations of the particular theory or dialect I propose (i.e., they would, and necessarily so, be equally true of any other proposed architecture).

This is one reason, among many, why I view 3-Lisp not *as* the

such as providing support for a routine to run exactly once per second), unless X is also real-time. So, to adopt a convenient way of speaking, I would say that being real-time "cross implementation boundaries downwards" (that is: that from a system's being real-time, one can conclude that the system on or in which it is implemented is also real-time—and hence all such systems below it, down to the hardware). Conversely, "being a finite state machine" is a property that crosses implementation boundaries *upwards*, since there is no way to implement a machine with an indefinitely unbounded store on top of one that has no such store. Needless to say, it does not cross implementation boundaries *downwards*; you can perfectly well implement a finite state machine in Lisp, which is not one.

The present point is that semantical properties in general—and thus reflection in particular—*do not cross implementation boundaries in either direction*. From neither X nor Y's being reflective, in the above example, can one deduce anything about whether the other is reflective.

For further discussion see «Ref aos».

contribution made in this dissertation, but rather as an *example to exhibit* its contribution: the conceptual structure of how to design and build a reflective architecture. Thus it is my hope that what would be useful from this dissertation for Doyle, or for anyone else in a parallel circumstance, is the detailed structure of a reflective system that I have attempted to explicate here—an architecture and a concomitant set of theoretical terms *to help such a person analyse and structure whatever architecture they design, adopt, or embrace*. Thus I would count the present contribution a success if it proved useful, for Doyle or anyone else, to make use of:

1. The φ/ψ distinction;
2. The relationship between semantical levels and reflective levels;
3. The encoding of the reflective model within the calculus;
4. The strategy of adopting a virtually infinite tower of processors as a finite model for level-shifting;
5. The semantic flatness and uniformity of a normalising processor;
6. The elegance of category-alignment;

And so forth. It is in this sense that I hope that the theory and understanding that 3-Lisp embodies will contribute to problem-solving research (and to programming language research), rather than the particular formalism I have developed and demonstrated by way of illustration.[x]

[x] As mentioned in the commentary included at the beginning of the POPL paper «ref», I believe it is fair to say that these hopes were entirely in vain. Dan Friedman, of Indiana University, was one of the most enthusiastic proponents of reflection in the programming language community; I owe him a debt of gratitude for the enthusiasm and support he offered subsequent of the publication of the POPL paper introducing 3-Lisp (reproduced here as chapter ■■). However as perhaps best illustrated in his own paper with Mitchell Wand (Friedman & Wand, 1984), the first thing that most people did, in bringing reflection into their own work, was to dismiss every one of these six claims.

For some of the reasons for this dismissal see the discussion cited above. Fundamentally I believe that it stems from a lack of theoretical concourse between the representational tradition (logic, data bases, knowledge rep-

The second type of research with which this project has strong ties is the general tradition of providing formalisms to be used as languages and vehicles for a variety of other projects—including the formal statement of theories, the construction of computational processes, the analysis of human language, and so forth. I take this tradition to be sufficiently broad (in particular, to include logic and the λ-calculus, plus virtually all programming language research) that it is difficult to say very much that is specific, though a few comments can be made.

First, I of course owe a tremendous debt to the Lisp tradition in general,[29] and also to the recent work of Steele and Sussman.[30] Particularly important is their Scheme dialect—in many ways the most direct precursor of 2-Lisp.[31] Second, my explicit attempt to unify the declarative and procedural aspects of this tradition has already been mentioned—a project that is (as far as I know) without precedent. Note, as mentioned in the Introduction, that I do not consider PROLOG[32] to count as having done this, since it provides two calculi together, rather than presenting a single calculus under a unified theory. Finally, as documented throughout the text, inchoate reflective behaviour can be found in virtually all comers of computational practice; the Smalltalk language,[33] to mention just one example, includes a meta-level debugging system which allows for the inspection and incremental modification of code in the midst of a computation.

The third and fourth classes of previous work listed above have to do with general semantics and with self-reference. The first of these is considered explicitly in chapter 3, where I compare my

resentation languages, etc., and the programming language community).

[29] References to specific Lisp dialects arc given in note ■■, above; more general accounts may be found in Allen (1978), Weisman (1967), Winston and Horn (1981), Charniak et al. (1980), McCarthy et al. (1965), and McCarthy and Talbott (forthcoming).

[30] Steele (1976), Steele & Sussman (1976, 1978b).

[31] In an early version of the dissertation I called Scheme "1.7-Lisp," since it takes what I see as approximately half of the step from Lisp 1.6 to the semantically rationalised 2-Lisp.

[32] Clark and McCabe (1979), Roussel (1975), and Warren et al. (1977).

[33] Goldberg (1981); Ingalls (1978).

approach to this subject with model theories in logic, semantics of the λ-calculus, and the tradition of programming language semantics; no additional comment is required here. Similarly, the relationship between the notion of reflection I present and traditional concepts of self-reference are taken up in more detail in chapter 5;[x] here I merely comment that my concerns, perhaps surprisingly, are constrained almost entirely to computational formalisms. Unless a formal system embodies a *locus of active agency*—an internal processor (i.e., process) of some sort—the entire question of causal relationship between an encoding of self-referential theory and what I consider a genuine reflective model cannot even be asked.

We often informally think of a natural deduction "process" or some other kind of deductive apparatus making inferences over first-order sentences, as a heuristic in terms of which to make sense of the formal notion of derivability. Strictly speaking, however, in the purely declarative tradition *derivability* is no more than a *formal relationship that holds between certain sentence types*; no activity is involved. There are no notions of *next* or of *when* a certain deduction is made. *If* one were to specify an active deductive process over such first-order sentences, then it is imaginable that one could include sentences (relative to some axiomatisation of that deductive process) in such a way that the operations of the deductive process were appropriately controlled by those sentences (this is the suggestion explored briefly in §2b). The resulting machine, however—not merely in its reflective incarnation, but even prior to that, by including an active agency—cannot fairly be considered simply *logic,* but rather a full computational formalism of some sort.

Needless to say, I believe that a reflective version of such a descriptive system could be built.[34] My position with respect to such an image rests on two observations: (i) the result would be an inherently *computational* artefact, in virtue of the addition of independent agency, and (ii) 3-Lisp, although reflective, is not yet such a formalism, since it is purely procedural.

[x] See also my "Varieties of Self-Reference," included here as Chapter ■■.

[34] In fact it is my intent to develop just such an architecture in the future.

I conclude with one final comparison. The formalism closest in spirit to 3-Lisp is Richard Weyhrauch's FOL system,[35] although my project differs from his in several important technical ways. First, like Doyle's system, FOL is a problem solver: it embodies a theorem-prover, although it is possible (through the use of FOL's meta-levels) to give it guidance about the deduction process. In spite of those facilities, however, FOL is not a *programming language*. Furthermore, FOL adopts—in fact explicitly endorses—the distinction between declarative and procedural languages (first order logic and Lisp, in particular), using the procedural calculus as a *simulation structure* rather than as a descriptive or designational language. Weyhrauch claims that the power that emerges from combining—but maintaining as distinct—these "language-simulation-structure" pairs, as he calls them ("L-S pairs"), at each level in his meta hierarchy, is one of his primary contributions. It is my own claim, in contrast, that the greatest power will arise from *dismantling* the difference between procedural and declarative calculi.

There are other differences as well. I take the interpretation function that maps terms onto objects in the world outside the computational systems (φ) to be foundational. It would appear in Weyhrauch's systems as if that particular semantical relationship is abandoned in favour of internal relationships between one formal system and another. A more crucial distinction is hard to imagine—though there is some evidence[36] that this apparent difference may have to do with our respective uses of terminology, rather than with deep ontological or epistemological beliefs.

In sum, FOL and 3-Lisp are technically quite distinct, and the theoretical analyses on which they are based almost unrelated. At a more abstract level, however, they are clearly based on similar—and perhaps parallel, if not identical—intuitions. Furthermore, I would argue that 3-Lisp represents merely a first step in the development of a fully reflective calculus based on a fully integrated theory of computation and representation; how such an eventual system, once it were defined, would differ from FOL remains to be

[35] Weyhrauch (1978).

[36] I am indebted to Richard Weyhrauch for personal communication on these points.

seen. It seems likely that the resulting unified calculus, rather than the dual-calculus nature, would be the most obvious technical distinction, although the actual structure of the descriptive language, semantical meta-theories, and so forth, are also likely to differ both in substance and in detail.

One remaining difference is worth exploring in part because it reveals a deep but possibly distinctive character of my treatment of Lisp. It is clear from Weyhrauch's system that he considers the procedural formalism to represent a kind of *model* of the world—in the sense of an (abstract) artefact whose structure or behaviour mimics that of some other world of interest. Under this approach the computational behaviour can be taken *in lieu of* or *in place of* the real behaviour in the world being studied. Consider for example the numeral addition that is the best approximation a computer can make to actually "adding numbers" (whatever that might be). When we type '`(+ 1 2)`' into a Lisp processor, and it returns '`3`', we are liable to take those numerals not so much as *designators* of the respective numbers, but instead as *models*. There is no doubt that the input expression '`(+ 1 2)`' is a linguistic artefact; on the view I will adopt in this dissertation there is no doubt that the resultant numeral '`3`' is also a linguistic artefact. I do want to admit, however, that there is a not unnatural tendency to think of the latter as "standing in place of" the actual number, in a different sense from standard designation or naming. It is this sense of *simulation* rather than *description* that, as far as I understand it, underlies Weyhrauch's use of Lisp.

I fundamentally believe that this is a limited view, however—and go to considerable trouble to maintain an approach in which all computational structures are taken to be semantical in something like a linguistic sense, rather than (being taken as) serving as models. Many issues are involved—having to do with such issues as truth, completeness, and so forth—that a simulation stance cannot deal with. At worst, moreover, adopting a simulation stance can lead to a view of computational models that runs in danger of being either radically solipsistic or even, I believe, nihilist. It is exactly the *connection* between a computational system and the world that motivates my entire approach; a connection that I believe can be ignored only at considerable peril. I in no way rule out computations that in different respects mimic the behaviour of the world they are about; it is clear that certain forms of human analysis involve just this kind of thinking ("stepping through" the transitions of some mechanism in one's head, for example, to "be sure that one understands it"). My point is only that such simulation is *still a kind of thinking about the world*; it is not the world being thought about.[x]

6b The Mathematical Meta-Language

Throughout the dissertation I will employ an informal meta-language, built up from a rather eclectic combination of devices from quantificational logic, the lambda calculus, and lattice theory, extended with some straightforward conventions (such as expressions of the form "if P then A else B" as an abbreviation for "$[P \supset A] \wedge [\neg P \supset B]$"). Notationally I will use set-theoretic devices (union, membership, etc.), but these should be understood as de-

[x] It was not until 1987 that Rodney Brooks first made his famous statement that the "representation" should be discarded in Artificial Intelligence systems—in favour of a view that, in his words, treated "the world as its own best model" (Brooks 1987); see also his "Intelligence Without Reason" and "Intelligence Without Representation" (Brooks 1991a & 1991b).

What I take to be significant about the widely-heralded "sea-change" to which Brooks' and others work led is the fact that it betrays what I am here attributing to Weyhrauch: a somehow tacit but deep assumption that "representation" meant constructing within the machine *a replica of the world as a whole*, which could be used in its place—as opposed to what cognitive scientists and philosophers of mind take a representational theory of mind to involve, which is that a person "represents" the world only in the sense of employing some interpreted symbols or structures with semantic content involving facts, entities, and states of affairs in the world. Even an internal structure with content along the lines of "Make sure you look out constantly and check intersection to make sure that it is empty!" would count as a representation on the latter, but apparently not the former, view.

It is hardly surprising that the "full simulation" view of representation needed to be eschewed—though to take that as a rejection of representation altogether is both an extreme and a binaristic reaction. Brooks later softened his view, saying that AI systems should use representation "only when necessary"—which opens the door to what representation had originally meant. For more on Brooks, what the circumstances are in which "representation is necessary," etc., see my "Rehabilitating Representation," included «ref; second volume?»

fined over *domains* in the Scott-theoretic sense, rather than over unstructured sets. The notations should by and large be self-explanatory; a few standard conventions worth noting are these:

1. ‘$[A \rightarrow B\ ]$’ refers to the domain of continuous functions from A to B;
2. ‘$F : [A \rightarrow B]$’ means that F is a function whose domain is A and whose range is B;
3. ‘$<S_1, S_2, \ldots S_k>$’ designates the mathematical sequence consisting of the designata of “S_1”, “S_2”, … “S_k”;
4. $^{\text{‘S1’}}$ refers to the i’th element of S, assuming that S is a sequence (thus $<A, B, C>^2$ is B);
5. ‘$[S \times R\ ]$’ designates the (potentially infinite) set of all tuples whose first member is an element of S and whose second member is an element of R;
6. ‘A^*’ refers to the power domain of A:
$$[\ A \cup [A \times A] \cup [A \times A \times A] \cup \ldots\ ]$$
7. Parentheses and brackets are used interchangeably to indicate scope and function application in the standard way.
8. Standard currying is employed to deal with functions of several arguments. Thus:

$\lambda A_1, A_2, \ldots A_k\ .\ E$	means	$\lambda A_1 . [\lambda A_2 . [\ldots\ .\ [\lambda A_k\ .\ E] \ldots]]$
$\lambda <A_1, A_2, \ldots A_k>\ .\ E$	means	$\lambda A_1 . [\lambda A_2 . [\ldots\ .\ [\lambda A_k\ .\ E] \ldots]]$
$F(B_1, B_2, \ldots B_k)$	means	$((\ldots((F(B_1))B_2)\ldots)B_k)$

If I wanted to be more precise, I would be stricter about the use of domains rather than sets, in order that function continuity be maintained, and so forth. It is not my intent here to make the mathematics rigorous, but I trust that it would be straightforward, given the accounts I set down, to take this extra step towards formal adequacy.

6c Examples and Implementations

A considerable number of examples are presented throughout the dissertation, which can be approximately divided into two groups: (i) formal statements about Lisp and about semantics, expressed in the meta-language; and (ii) illustrative programs and structures expressed in Lisp itself (most of the latter are in one of the three Lisp dialects, though a few are in standard dialects as well). As the preceding discussion suggests, the meta-linguistic characterisations have not been checked by formal means for consistency or accuracy; the proofs and derivations were generated by the author using paper and pencil. The program examples, on the other hand, were all tested on computer implementations of 1-Lisp, 2-Lisp, and 3-Lisp developed in the MacLISP and "Lisp Machine" Lisp dialects of Lisp at MIT.[37] Thus, although the examples in the text were typed in by the author as text—i.e., the lines of characters in this document are not actual photocopies of computer interaction—each was nevertheless verified by these implementations. However the implementation presented in the Appendix is a photocopy of the actual computer program listing. Any residual errors (it is hard to imagine every one has been eliminated) must have arisen either from typing errors or from mistakes in the implementation itself.[x]

[37] A complete program listing of the third of these—a MacLISP implementation of 3-Lisp —is given in the Appendix.

[x] This dissertation was written in 1981 on the Xerox Alto minicomputer—arguably the first "personal computer"—developed at the Xerox Palo Alto Research Center (PARC) in the 1960s. It used Bravo, the first "WYSIWYG" ("what you see is what you get") document preparation system. The 3-Lisp implementation was developed in MacLISP , a dialect of Lisp implemented under "ITS" ("Incompatible Time-Sharing System") at the Artificial Intelligence Laboratory at MIT, running on Digital Equipment Corporation PDP-6 and PDP-10.

References (original)

Allen, Jon, *Anatomy of LISP*. New York: McGraw-Hill (1978).

Bobrow Daniel G., (ed.) *Artificial Intelligence,* 13:1,2 (Special Issue on Non-Monotonic Reasoning), (1980).

Bobrow, Daniel G., and Winograd, Terry, "An Overview of KRL: A Knowledge Representation Language," *Cognitive Science* 1:3–46 (1977)

Bobrow, Daniel G., Winograd, Terry et al., "Experience with KRL-O: One Cycle of a Knowledge Representation Language," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* Cambridge, Mass (August 1977) pp. 213–22.

Bobrow, Daniel G., and Wegbreit, Ben, "A Model and Stack Implementation of Multiple Environments," *Communications of the* ACM 16, 10:591–603 (Oct 1973).

Brachman, Ronald, "Recent Advances in Representation Languages," invited presentation at the First Annual National Conference on Artificial Intelligence, Stanford, California, (August 1980), sponsored by the American Association for Artificial Intelligence.

Brachman, Ronald and Smith, Brian Cantwell, (eds.), *Special Issue on Knowledge Representation,* SIGART *Newsletter,* 70 (February 1980).

Charniak, Edward, Riesbeck, Chris, and McDermott, Drew, *Artificial Intelligence Programming,* Hillsdale, NJ: Lawrence Erlbaum (1980).

Church, Alonzo, *The Calculi of Lambda-conversion,* Annals of Mathematics Studies 6, Princeton, NJ: Princeton University Press (1941).

Clark, K.L., McCabe F. (1979). *Programmer's guide to* IC-*Prolog.* CCD Report 79/7, London: Imperial College, University of London.

Davis, R. "Applications of Meta Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases," PhD thesis, Stanford University, Stanford, California; also in Davis, R., and Lenat, D., (eds.), *Knowledge-Based Systems in Artificial Intelligence,* New York: McGraw-Hill (1980a).

———, "Meta-Rules: Reasoning about Control", M.I.T. Artificial Intelligence Laboratory Memo AIM-576 (1980b); also *Artificial Intelligence* 15:3, December 1980, pp. 179–222.

deKleer, Johan, Doyle, Jon, Steele, Guy L. Jr., and Sussman, Gerald J., "Explicit Control of Reasoning," *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages,* Rochester, N.Y. (1977); also M.I.T. Artificial Intelligence Laboratory Memo AIM-427 (1977).

Donnellan, Kennett, "Reference and Definite Descriptions," *Philosophical Review* 75:3 (1966), pp. 281–304.; reprinted in Rosenberg and Travis (eds.), *Readings in the Philosophy of Language,* Prentice-Hall (1971).

Doyle, Jon, "A Truth-Maintenance System," *Artificial Intelligence* 12:231–272 (1979).

———, *A Model for Deliberation, Action, and Introspection,* doctoral dissertation submitted to the Massachusetts Institute of Technology; also M.I.T. Artificial Intelligence Laboratory Memo AIM-TR-581 (1980).

Dreyfus, Hubert, *What Computers Can't Do*, New York: Harper and Row (1972).

Fodor, Jerry, *The Language of Thought*, New York: Thomas Y. Crowell, Company (1975): paperback version, Cambridge: Harvard University Press, 1979.

———, "Tom Swift and his Procedural Grandmother," *Cognition* 6 (1978); reprinted in Fodor, Jerry, *Representations*, Cambridge: Bradford, 1981.

———, "Methodological Solipsism Considered as a Research Strategy in Cognitive Psychology," *The Behavioral and Brain Sciences* 3:1 (1980) pp. 63–73; reprinted in John Haugeland (ed.), *Mind Design*, Cambridge: Bradford, 1981, and in Jerry Fodor, *Representations*, Cambridge: Bradford 1981.

———, *The Modularity of Mind*, Cambridge: Bradford (forthcoming).

Frege, Gottlob, *Die Grundlagen der Arithmetik: Eine logisch-mathematische Untersuchung über den Begriff der Zahl* (Breslau, 1884); reprinted in *The Foundations of Arithmetic, A logico-mathematical Inquiry into the Concept of Number*, English translation by John L. Austin, Evanston, IL: Northwestern University Press (1950).

Galley, S. W., and Pfister, G., *The* MDL *Language*, Programming Technology Division Document SYS.11.01. Laboratory of Computer Science, M.I.T. (1975).

Genesereth, Michael and Lenat, Douglas B. "Self-Description and Modification in a Knowledge Representation Language," Report of the Heuristic Programming Project of the Stanford University Computcr Science Dept., HPP-80-10 (1980).

Goldberg, Adele et al. "Introducing the Smalltalk-80 System," and other Smalltalk papers, *Byte* 6:8, (August 1981).

Gordon, Michael J. C., "Models of Pure LISP," Dept. of Machine Intelligence, Experimental Programming Reports No. 30, University of Edinburgh (1973).

———, Operational Reasoning and Denotational Semantics," Stanford University Computer Science Dept. Deport No. STAN-CS-75-506. (1975a)

———, "Toawards a Semantic Theory of Dynamic Binding," Stanford University Artificial Intelligence Laboratory, Memo 265, Stanford University Computer Science Dept. Report No. STAN-CS-75-507 (1975b).

———, *The Denotational Description of Programming Languages: An Introduction*, New York: Springer-Verlag (1979).

Greiner, R., and Lenat, D. B., "A Representation Language Language", *Proceedings of the First Annual National Conference on Artificial Intelligence*, Stanford Univ., (August 1980), pp. 165–169.

Haugeland, John, "The Nature and Plausibility of Cognitivism," *The Brain and Behavioral Sciences* 1 (1978).

Hayes, Patrick J., "In Defense of Logic," in *Proc. Fifth International Joint Conference on Artificial Intelligence,* Massachusetts Institute of Technology (August 1977) pp. 559–65; available from Carnegie-Mellon University, Pittsburgh. PA.

——— "The Naive Physics Manifesto", unpublished manuscript (May 1978).

———, Personal conversations on the GOLUM deduction system (1979).

Hewitt, Carl, "Description and Theoretical Analysis (using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot," MIT Artificial Intelligence Laboratory TR-258 (1972).

———, "Viewing Control Structures as Patterns of Passing Messages," *Artificial Intelligence,* 8:3. (June 1977) pp. 324–64.

Hewitt, Carl et al. "Behavioral Semantics of Non-recursive Control Structures," *Proc. Colloque sur la Programmation,* B. Robinet (ed.), in *Lecture Notes in Computer Science,* 19, pp. 385–407 Berlin: Springer-Verlag (1974).

Ingalls, Daniel H. "The Smalltalk-76 Programming System: Design and Implementation," *Conference Record of the Fifth Annual Symposium on Principles of Programming Languages,* Tucson, Arizona (January 1978) pp. 9–16.

Kleene, Stephen, *Introduction to Metamathematics,* Princeton: D. Van Nostrand (1952).

Kowalski, Robert A., "Predicate Logic as a Programming Language," *Proceedings IFIP,* Amsterdam: North Holland (1974) pp. 569–74.

———, "Algorithm = Logic + Control", *CACM* (August 1979).

Kripke, Saul, "Outline of a Theory of Truth," *Journal of Philosophy,* 72:690–716 (1971).

Lewis, David, "General Semantics," in Davidson and Harman (eds.), *Semantics of Natural Languages,* Dordrecht, Holland: D. Reidel (1972), pp. 169–218.

Maturana, Humberto, and Varela, Francisco, *AutojJoietic SysletnS,* in Boston studies in the philosophy of science. Boston: D. Reidel, (1978); originally issued as B.C.L. Report 9.4, Biological Computer Laboratory, University of Illinois, 1975.

McAllester David A. "A Three-Valued Truth Maintenance System," MIT Artificial Intelligence Laboratory Memo AIM-473 (1978).

McCarthy, John, "Programs With Common Sense," in Marvin Minsky (ed.), *Semantic Information Processing.* Cambridge: MIT Press (1968), pp. 403–18.

McCarthy, John et al., *LISP 1.5 Programmer's Manual*, Cambridge, Mass.: MIT Press (1965).

McCarthy, John and Talbott, Carolyn, *LISP: Programming and Proving,* Cambridge, Mass.: Bradford (forthcoming).

McDermott, Drew, and Doyle, Jon, "Non-Monotonic Logic I," M.I.T. Artificial Intelligence Laboratory Memo AIM-486 (1978).

McDermott, Drew, and Sussman, Gerald, "The CONNIVER Reference Manual," M.I.T. Artificial Intelligence Laboratory Memo AIM-259a, Cambridge, Mass. (1973).

Minsky, Marvin, "Matter, Mind, and Models", in *Semantic Information Processing,* M. Minsky (ed.), Cambridge: MIT Press (1968).

———, "A Framework for the Representation of Knowledge", in P. Winston (ed.), *The Psychology of Computer Vision,* New York: McGraw-Hill (1975) pp. 211–77.

Montague, Richard, "The Proper Treatment of Quantification in Ordinary English," in J. Hintikka, J. Moravcvcsik, and P. Suppes (eds.), *Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics,* Dordrecht: Reidel (1973) pp. 221–42; reprinted in Thomason (1974).

———, "Pragmatics and Intensional Logic", *Synthese* 22 (1970) pp. 68–94; reprinted in R. H. Thomason (ed.), *Formal Philosophy: Selected Papers of Richard Montague,* New Haven: Yale Univ. Press, 1974.

Moon, David, "MacLISP Reference Manual", M.I.T. Laboratory tor Computer Science, Cambridge, Mass. (1974).

Moses, J., "The Function of FUNCTION in LISP," ACM SIGSAM Bulletin, pp. 13–27, (July 1970); also M.I.T. Artificial Intelligence Laboratory Memo AIM-199 (1970).

Newell, Allen, and Simon, Herbert, "The Logic Theory Machine: a Complex Information Processing System", *IEEE Transactions on Information Theory,* Vol. IT-2, No.3, pp. 61-79.

———. "GPS, a Program that Simulates Human Thought", in B. A. Feigenbaum and J. Feldman (eds.). *Computers and Thought,* New York: McGraw-Hill (1963).

Nilsson, Nils, "Artificial Intelligence: Engineering, Science, or Slogan?" manuscript (to be published), (April 1981).

Pitman, Ken, "Special Forms in LISP", *Conference Record of the 1980 LISP Conference,* Stanford University (August 1980), pp. 179–87.

Quine. Willard von Orman, *Mathematical Logic,* [New York: Norton, 1940], Cambridge: Harvard University Press, 1947; revised edition, Cambridge, Harvard University Press (1951).

———, "Identity, Ostension, and Hypostasis," in *From a Logical Point of View,* Cambridge: Harvard University Press, (1953a); reprinted in paperback by Harper Torchbooks, 1963.

———, "On What There Is," in Quine, W. V. 0., *From a Logical Point of View,* Cambridge: Harvard University Press, (1953b); reprinted in paperback by Harper Torchbooks, 1963.

———, "Three Grades of Modal Involvement", in *The Ways of Paradox,* *and Other Essays*, Cambridge: Harvard Univ. Press (1966). Quine, W. V. 0., and Ullian, J. S., *The Web of Belief*, New York: Random House (1978).

Reiter, Ray, "On Reasoning by Default," *Proc. Second Conference on Theoretical Issues in Natural Language Processing*, University of Illinois at Champaign-Urbana (1978) pp. 210–18.

Rogers, Hartley Jr., *Theory of Recursive Functions and Effective Computability*, New York: McGraw-Hill (1967).

Roussel, P., "PROLOG: Manuel de Reference et d'Utilisation", Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy (1975).

Russell, Bertrand, "Mathematical Logic as Based on the Theory of Types," *American Journal of Mathematics* 30:222–62 (1908); reprinted in Van Heijenoort, J. (ed), *From Frege to Gödel: A Source Book in Mathematical Logic, 1*879-1931, Cambridge, Mass.: Harvard (1967).

Searle, John R., *Speech Acts: An Essay in the Philosophy of Language*, Cambridge: Cambridge Univ. Press (1969).

———, "Minds, Brains, and Programs," *The Behavioral and Brain Sciences* 3:3 (1980) pp. 417–57; reprinted in John Haugeland (ed.), *Mind Design*, Cambridge: Bradford 1981, pp. 282-306.

Stallman, Richard M., and Sussman, Gerald J., "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," *Artificial Intelligence* 9:2 (1977) pp. 135–96; also in *Artificial Intelligence: An MIT Perspective*, Volume 1, Patrick H. Winston and Richard H. Brown (eds.), pp. 31-91, Cambridge: M.I.T. Press (1979).

Steele, Guy, "LAMBDA: The Ultimate Declarative," M.I.T. Artificial Intelligence Laboratory Memo AIM-379 (1976).

———, "The Definition and Implementation of a Computer Programming Language Based on Constraints," Ph.D. Dissertation, M.LT. Artificial Intelligence Laboratory, Report AIM-595 (1980).

Steele, Guy and Sussman, Gerald. "LAMBDA: The Ultimate Imperative," M.LT. Artificial Intelligence Laboratory Memo AIM-353 (1976).

———, "The Revised Report on SCHEME, A Dialect of LISP", M.I.T. Artificial Intelligence Laboratory Memo AIM-452 (1978a).

———, "The Art of the Interpreter, or, The Modularity Complex (Parts Zero, One, and Two)", M..I.T. Artificial Intelligence Laboratory Memo AIM-453, Cambridge, Mass. (1978b).

———, "Constraints", M.I.T. Artificial Intelligence Laboratory Memo AIM-502 (1979).

Stefik, Mark J., "Planning with Constraints (MOLGEN, Part 1)", *Artificial Intelligence* 16:2 (July 1981a) pp. 111–39.

———, "Planning and Meta-Planning (MOLGEN: Part 2)", *Artificial Intelligence* 16:2 (July 1981b) pp. 141–69.

Stoy, Joseph, *Denotational Semantics: The Scott-Strachey Approach to Pro-*

gramming Language Theory, Cambridge: MIT Press (1977).

Sussman, Gerald, and Steele, Guy, "SCHEME: An Interpreter for Extended Lambda Calculus," M.I.T. Artificial Intelligence Laboratory Memo AIM-349 (1975).

———, "CONSTRAINTS: A Language for Expressing Almost-Hierarchical Descriptions," *Artificial Intelligence* 14:1 (August 1980) pp. 1-39.

Sussman, Gerald, et al. "Micro-PLANNER Reference Manual" M.I.T. Artificial Intelligence Laboratory Memo AIM-203a (1971).

Tarski, Alfred, "The Concept of Truth in Formalized Languages" (1936), in Tarski, Alfred, *Logic, Semantics, Metamathematics*, Oxford (1956).

———, "The Semantic Conception of Truth and the Foundations of Semantics". *Philosophical and Phenomenological Research* 4:341–76 (1944); reprinted in Linksy (ed.), *Semantics and the Philosophy of Language*, Urbana: University of Illinois, 1952, pp. 13-47.

Teitelman, Warren, "InterLISP Reference Manual," Palo Alto: Xerox Palo Alto Research Center (1978).

Tennent, R.D., "The Denotational Semantics of Programming Languages", *Communication of the* ACM 19:8 pp. 437-453 (Aug. 1976).

Thomason, Rich, (ed.), *Formal Philosophy: Selected Papers of Richard Montague*, New Haven: Yale University Press (1974.)

Warren, David, Pereira, Luis, and Pereira, Fernando, "PROLOG: The Language and its Implementation Compared with LISP", *Proc. Symposium on AI and Programming Languages*, Rochester, New York, ACM SIGPLAN/SIGART *Notices*, 12:8 (August 1977) pp. 109-115.

Weissman, Clark, *LISP* 1.5 *Primer*, Belmont: Dickenson Press (1967).

Weinreb, Dan, and Moon, David, *LISP Machine Manual*, Cambridge: Massachusetts Institute of Technology (1981).

Weyhrauch, Richard W., "Prolegomena to a Theory of Mechanized Formal Reasoning", Stanford University Artificial Intelligence Laboratory, Memo AIM-315 (1978); °also *Artificial Intelligence* 13:1.2 (1980) pp. 133-170.

White, John L., "NIL - A Perspective", *Proceedings of the* MACSYMA *Users' Conference*, Washington, D. C. pp. 190-199 (June 1979). Available from The Laboratory for Computer Science, M.I.T., Cambridge, Mass.

Winograd, Terry, *Understanding Natural Language*, Academic Press (1972).

———, "Frame Representation and the Declarative-Procedural Controversy," in D. G. Bobrow and A. Collins, (eds.), *Representation and Understanding: Studies in Cognitive Science*, New York: Academic Press (1975) pp. 185–210.

Winston, Patrick H and Horn, Bertold K. P., *LISP*, Reading, Mass: AddisonWesley (1981).

References (added)

Brooks, Rodney A., “Intelligence Without Representation,” *Preprints of the Workshop in Foundations of Artificial Intelligence*, Endicott House, Dedham, MA, June, 1987; final version published in *Artificial Intelligence Journal* (47), 1991a, pp. 139–159.

———, “Intelligence Without Reason”, in the *Proceedings of 12th international Joint Conference on Artificial Intelligence*, Sydney, Australia, August 1991b, pp. 569–95.

Dennett, Daniel, The Intentional Stance, Cambridge, Mass.: MIT Press, 1987.

Dennett, Daniel … on original/derivative etc.

Dixon, Mike … on Amala

Fodor, Jerry A., “Methodological solipsism considered as a research strategy in cognitive psychology”, *Behavioral and Brain Sciences*, 3, 1980; pp. 63-73. Reprinted in Rosenthal, D., ed., *The Nature of Mind*, Oxford: Oxford University Press, 1990.

Friedman, Daniel P. and Wand, Mitchell, “Reification: Reflection without metaphysics,” Proceedings of the 1984 ACM Symposium on LISP and functional programming, Austin, Texas, United States, pp. 348–55).

Haugeland, John … on original/derivative etc.

Searle, John … on original/derivative etc.

3 — Reflection and Semantics in LISP

Brian Cantwell Smith*
University of Toronto

1 Abstract

A general architecture is presented, called **procedural reflection**, designed to support self-directed reasoning in a serial programming language. The architecture, illustrated in a revamped dialect of Lisp called **3-Lisp**, involves three steps: (i) reconstructing the semantics of a language so as to deal with both declarative and procedural aspects of program meaning; (ii) embedding a theory of the language—including of its semantics—within the language; and (iii) defining an infinite tower of procedural self-models in terms of this embedded theory, very much like a tower of metacircular interpreters, except causally-connected to each other in a simple but crucial way. In a procedurally reflective architecture, any aspect of process state that can be described in terms of the theory can be rendered explicit, in

Last edited: December 17, 2009
Draft only (version 0.80) Please do not copy or cite.
Comments welcome brian.cantwell.smith@utoronto.ca

Edited version of a paper first published in the *Conference Record of the Elev-enth Annual ACM Symposium on Principles of Programming Languages* (POPL), Salt Lake City, Utah, Jan. 1984, pp. 23–35. The original version was also published as Report No. CSLI-84-8, Stanford University Center for the Study of Language and Information, July 1984.

[*]Faculty of Information, University of Toronto, 90 Wellesley St W, Toronto, Ontario M5S 1C5 Canada.

structures accessible for program examination and manipulation. Procedural reflection enables a user to define complex programming constructs by writing, within the programming language, direct analogues of those metalinguistic semantical expressions that would normally be used to describe them.

It is argued that the concept of procedural reflection should be added to any language designer's tool kit.

2010 Perspective[⊠1]

The work reported here, on procedural reflection and 3-Lisp, started out as what I expected to be a small design study—part of a (hopelessly ambitious) project I had undertaken, as a graduate student in the 1970s, to develop a fully reflective knowledge representation system. That project, to have been called Mantiq,[⊠2] never saw the light of day, most pointedly due to my encounter with the fundamental inability of Artificial Intelligence and computer science to deal adequately with the challenges of real-world ontology (the nature of objects, ambiguity and vagueness, relationality and process, etc.). But there were other challenges as well: another goal was to define the Mantiq structural field (effectively: its object or memory system—see p. ■■) at a sufficiently high level of abstraction so as to be able to "fuse" meta-structural and intensional identity, so that *structural* identity could be identified with (and thus used to determine) identity of *meaning*. [⊠3] The idea was to employ a computationally-intensive background relaxation algorithm to implement the "structural field" (memory system), loosening operational identity criteria to the point that, for example, the Mantiq analogues of (⊠x,y . x+y) and (⊠a,b . b+a) would appear to be structurally indistinguishable.

I still think that this issue of intensional identification would be a worthwhile goal to pursue, especially since processing power today would make approximating it more computationally feasible than it was thirty years ago. At any rate, against this background of unrealistic dreams, the 3-Lisp project[⊠4] was intended as a site to work out the design details of reflection's self-referential structure. In particular, the idea of understanding level-shifting in terms of an idealized unbounded "tower" of referential layers struck me then (and still does now) as at least a good initial idea about the structure of reflection.[⊠5] So I set out to explore it within the familiar context of Lisp, the "lingua franca" programming language

1 Introduction

Among programming languages, Lisp is famous for (among other things) providing inchoate self-referential capabilities: standard coding of programs as data structures (s-expressions), a primitive quotation function (QUOTE), explicit access to interpreter procedures (EVAL and APPLY), support for meta-circular interpreters, etc. Yet these capacities have not led to a general understanding of what it is for a computational system to reason, in substantial ways, about its own operations and structures.

There are several reasons we have not developed such an account. First, there is more to reasoning than reference; one also needs a theory, in terms of which to make sense of the referenced domain. A computer system able to reason about itself—what I will call a **reflective** system—will therefore need an account of itself embedded within it. Second, there must be a systematic, causally effective relationship between that embedded account and the system it describes. Without such a connection, the account would be useless—as disconnected as the words of a hapless drunk who carries on about the evils of inebriation, without realising that his story applies to himself. Traditional language embeddings in Lisp (meta-circular interpreters and implementations of other languages) are inadequate in just this way; they provide no means for the implicit state of the Lisp process to be reflected, moment by moment, in the explicit terms of the embedded account. Third, a reflective system must be given an appropriate vantage point at which to stand, far enough away to have itself in focus, and yet close enough to see the important details.

This paper presents a general architecture, called **procedural reflection**, to support self-directed reasoning in a serial programming language. The architecture, illustrated in a revamped Lisp dialect called **3-Lisp**, solves all three problems with a single mechanism. The basic idea is to define an infinite tower of procedural self-models, very much like metacircular interpreters,[1] except connected to each other in a simple but critical way. In such an architecture, any aspect of a process' state that can be described in terms of the theory can be rendered explicit, in program accessible structures, at an arbitrary points throughout a computation. Furthermore, as I

[1] Steele and Sussman (1978b).

will demonstrate, this apparently infinite architecture can be finitely and efficiently implemented.

The architecture allows the user to define complex programming constructs (such as escape operators, deviant variable passing protocols, and debugging primitives) by writing, within the language, direct analogues of the metalinguistic semantical expressions that would normally be used to describe them. As is always true in semantics, the metatheoretic descriptions must be phrased in terms of some particular set of concepts; in the 3-Lisp case I use a theory based on *environments* and *continuations*. A 3-Lisp program, therefore, at any point during a computation, can easily obtain representations of the environment and continuation characterising the state of the computation at that point. As a result, such constructs as THROW and CATCH, which must otherwise be provided primitively, can be easily defined in 3-Lisp as user procedures (and defined, furthermore, in code that is almost isomorphic to the ⊠-calculus equations one normally writes, in the metalanguage, to describe such constructs). Moreover, these and other analogous control constructs can be defined without having to write the entire program in a continuation-passing style, of the sort illustrated in Steele (1976).

The point is not to decide at the outset what should and what should not be explicit, in other words (in Steele's example, continuations must be passed around explicitly from the beginning).[a] Rather,

[a] *(Note: footnotes indicated with letters rather than numerals, and sans-serif font, as in this case, are annotative notes added in 2010, rather than material that appeared in the original paper.)*

This phrasing is somewhat disingenuous, since in a procedurally reflective dialect of the sort presented here the language designer must decide, advance, what aspects of the language *will be able to be made explicit* to user code; those aspects must then be dealt with, explicitly, in the metatheory in terms of which the reflective processor and dialect are themselves defined, and then provided for in the implementation. The original paper would have been better phrased if written as follows: "Although the metatheory (and reflective processor) must deal explicitly with all of those aspects of the language that can, at any point, be made explicit, any user code that does not want to deal with them need not deal with them explicitly. In Steele's dialect, in contrast, in order for an aspect to be referred to explicitly at any point, it must be explicit *throughout the program*. In a sense, therefore, reflection can be understood as providing something like contextual information hiding—or perhaps more the reflective architecture provides a method of making some aspects of the computation explicit, right in the midst of a computation, even if they were implicit a moment earlier—and in such a way that they can be made implicit once again, a moment later. It provides a mechanism, in other words, when circumstances warrant it, of stepping back, "pulling information out of the sky," dealing with that information appropriately, and then returning into the regular implicit flow of the program.

The thesis on which the 3-Lisp definition rests is the following:

Reflection is simple to build on a semantically sound base. **[R]**

By "semantically sound" I mean more than that the semantics be carefully formulated. Rather, it is assumed throughout that computational structures have a semantic significance that transcends their behavioural import—or, to put this another way, that programs and computational structures are *about* something, over and above the *causal effects* they have on the systems they inhabit. Lisp's NIL, for example, evaluates to itself forever—that is its procedural impact. In addition, however, in some contexts—and partially independently—it also *stands for falsehood*. It is that sense of "*meaning false*" that I take to be its declarative import. To be considered "semantically sound," a reconstruction of Lisp semantics must deal explicitly with both of these dimensions of the overall significance of computational structures—both procedural and declarative.[2]

In what follows I will use the phrases "*procedural result*" (or "*what it returns*") to name that to which its effective treatment gives rise, and "*declarative import*" for what a structure designates, declaratively. As well as distinguishing result and import, I will also discriminate

[2] This distinction between the procedural and declarative aspects of a program's meaning differs from the traditional distinction in programming language theory between operational and denotational semantics. It is a reconstruction developed within a view that programming languages are properly to be understood in the same theoretical terms used to understand natural language and mind—not just other computer languages.

accurately, *contextually-dependent explicitization of otherwise implicit information.*"

The next sentence in the text is more accurate, and more useful.

entities, such as numerals and numbers, that are isomorphic but not identical, if they differ in respect of either import or result.[3] Both distinctions are instances of the general intellectual hygiene of avoiding use/mention errors. Lisp's basic notion of *evaluation*, I will argue, is fundamentally confused on both counts—and should be replaced with independent notions of **designation** and **simplification**. The result will be illustrated in a semantically rationalised dialect, called **2-Lisp**, based on a *simplifying* (designation-preserving) term-reducing processor.

The practical import of thesis [R] is demonstrated in a two-stage argument:

1. The semantically rationalised 2-Lisp is more elegant and theoretically cleaner than any prior Lisp dialect (including both Lisp 1.5 and Scheme); and

2. The reflective dialect 3-Lisp can be very simply defined on top of 2-Lisp—whereas a reflective version of a non-semantically-rationalised Lisp dialect would be inelegant in a spate of ways: gratuitously challenging to design, architecturally baroque, and much more difficult to understand.

The strategy of presenting the general architecture of procedural reflection by developing a concrete instance of it was selected on the grounds that a genuine theory of reflection (perhaps analogous to the theory of recursion) would be difficult to motivate or defend without taking this first, more pragmatic, step. In section 10, however, I will sketch a general "recipe" for adding reflective capabilities to any serial language; 3-Lisp is the result of applying this conversion process to the non-reflective 2-Lisp.

It is sometimes said that there are only a few constructs from which programming languages are assembled—including, for example, predicates, terms, functions, composition, recursion, abstraction, a branching selector, and quantification. Though different from these notions (and not definable in terms of them), reflection is perhaps best viewed as a proposed addition to that family. Given this view, it is helpful to understand reflection by comparing it, in particu-

[3] Numerals denote numbers, but (at least in ordinary circumstances) numbers do not denote at all, not being *symbols*.

lar, with recursion—a construct with which it shares many features. Specifically, recursion can seem viciously circular to the uninitiated, and can easily lead to confused implementations if poorly understood. Careful theoretical analysis, however, backed by mathematical theory, underwrites our ability to use recursion in programming languages without doubting its fundamental soundness (in fact, for many programmers, without understanding much about the formal theory at all). Reflective systems, similarly, are initially likely to seem viciously circular (or at least infinite), and are correspondingly difficult to implement without an adequate understanding. The intent of this paper, however, is to argue that reflection is in fact as well-tamed a concept as recursion, and potentially as efficient to use. The long-range goal is not to force programmers to understand the intricacies of designing a reflective dialect, but rather to enable them to use reflection and recursion with equal abandon.

2 Motivating Intuitions

Before taking up technical details, it will help to layout some motivations and assumptions.

By 'reflection' in its most general sense, I mean the ability of an agent to reason not only introspectively, about its self and internal thought processes, but also externally, about its behaviour and situation in the world. Ordinary reasoning is external in a simple sense: most of what we think about (chairs, other people, bank accounts, houses, politics, etc.) is external to us. The point of reflection is to give an agent a more sophisticated stance *from which to consider its own presence in that embedding world*. There is a growing consensus[4] that reflective abilities underlie much of the plasticity with which we deal with the world, both in language (such as when one says "Do you understand what I mean?") and in thought (such as when one wonders how to be compassionate about delivering bad news). Common sense suggests that reflection enables us to master new skills, cope with incomplete knowledge, define terms, examine assumptions, review and distill experiences, learn from unexpected situations, plan, check for consistency, and recover from mistakes.

Although this paper focuses on reflection in programming

[4] See Doyle (1980), Weyrauch (1980), Genesereth and Lenat (1980), and Batali (1983).

languages, most of the driving intuitions on which it is based are grounded in considerations of human rationality and language. Tentative steps towards computational reflection, however, are emerging in computational practice, and have also had a motivating impact here. Debugging systems, trace packages, dynamic code optimizers, runtime compilers, macros, metacircular interpreters, error handlers, type declarations, escape operators, comments, and a variety of other programming constructs in one way or another involve structures that refer to or deal with other parts of a computational system. These practices suggest. as a first step towards a more general theory, defining a limited and rather introspective notion of "procedural reflection": self-referential behaviour in procedural languages, in which expressions are primarily used instructionally, to engender behaviour, rather than assertionally, to express judgments or make claims. It is the hope that the lessons learned in this smaller task will serve well in the larger account.[b]

I mentioned at the outset that the general task, in defining a reflective system, is to embed a theory of the system in the system in such a way as to support smooth shifting between reasoning directly about the world and reasoning about that reasoning. Because the subject matter is reasoning, moreover, not merely language, an additional requirement is placed on this embedded theory, also already mentioned, beyond its being descriptive and true: it must also be what I will call **causally connected**, so that the reflective accounts of objects, events and states of affairs are directly tied to those selfsame objects, events and states of affairs. This causal relationship must run both directions: from event to description, and from description back to event. The goal is almost that of creating a magic kingdom, where from a cake you can automatically obtain a recipe, and from a recipe automatically produce a cake.

[b] In part this is a reference to Mantiq, but I had also planned to develop a next dialect in the series, to be called "4-Lisp," which was to include semantically-rationalized data structures for (external) reference to the real-world, but otherwise to retain 3-Lisp's basic style and control structure. Like Mantiq, 4-Lisp never materialized, due to the challenges of developing representational regimens adequate to real-world ontology.

Existing logical and mathematical cases of self-reference, including both self-referential statements, and models of syntax and proof theory, involve no causation at all, since there is no temporality or behaviour (neither logical nor mathematical systems, per se, *run*). Effective causation is a critical part of any reflective agent, however. As a human example, suppose you were to capsize while canoeing through difficult rapids, and were to swim to shore to figure out what you did wrong. In terms of what I will call "upwards" causal connection, you would need a description of *what you were doing at the moment the mishap occurred*; in the concrete exigencies of that circumstance, merely having a name for yourself, or even a general description of yourself, would be useless. Similarly, in order for your on-shore reflections to be of any subsequent paddling use, you would need "downwards" causal connection as well; no good will come from your merely contemplating a disconnected theory of a wonderfully improved you. As well as stepping back and being able to think about your behaviour, in other words, you must also be able to "step forwards," as it were—to embrace a revised theory of self and "dive back in under it," adjusting your behaviour so as to satisfy the new account. And finally, as already mentioned, when you take the step backwards, to reflect, you need a place to stand that has just the right combination of connection and detachment to make this whole process effective and efficient (it is not an accident that the moment of self-contemplation is like to occur *on shore*).

Reflective computational systems, similarly, must provide both directions of causal connection, and an appropriate vantage point. For example, consider a debugging system that accesses stack frames and other implementation-dependent representations of processor state, in order to give the user an account of what a program is up to in the midst of a computation. Note, first, that stack-frames and implementation byte-codes really are just descriptions, in a rather inelegant language, of the state of the process they describe. Like any description, they make explicit some of what was implicit in the process itself (this is one reason they are useful in debugging). Furthermore, because of the nature of implementation—because, that is, they are constitutively enabling descriptions, not detached observations—they are always available in the implementing code, and always true. They have these properties because they play a causal role in the very existence of the process they implement, and therefore automatically solve the "reality-to-description" direction of causal connection. Second, debugging systems must solve the "description-to-reality" problem, by providing a way of making revised descriptions of the process true of that process. They carefully provide facilities for altering the underlying state, based on the user's description of what that state should be (i.e., "return from this stack frame immediately"). Without this "map to reality" direction of causal connection, the debugging system, like an abstract model, could have no effect on the process it was examining. And finally, programmers who write debugging systems wrestle with the problem of providing a proper vantage point. In this case, practice has been particularly atheoretical; it is typical to arrange, very cautiously, for the debugger to tiptoe around its own stack frames, in order to avoid control challenges, variable clashes and other unwanted interactions.

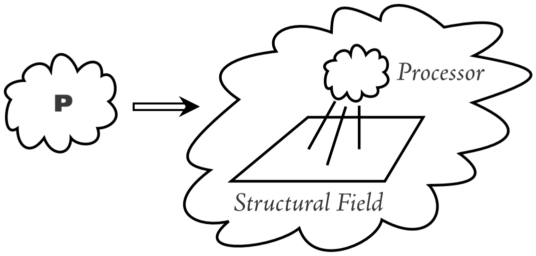


Figure 1 — A Serial Model of Computation

As will be evident in the design of 3-Lisp, all of these concerns can be dealt with in a reflective language in ways that are simple, theoretically elegant, and implementation-independent. The procedural code in the metacircular processor serves as the "theory" discussed above; the causal connection is provided by a mechanism whereby procedures at one level in the reflective tower are run in the process one level above (a clean way, essentially, of enabling a program to define subroutines to be run in its own implementation). In one sense it is all straightforward; the subtlety of 3-Lisp has to do not so much with the power of such a mechanism, which once presented is evident, but with *how such power can be finitely provided*—a question addressed in section 9.

Some final assumptions. I assume a simple serial model of computation, illustrated in figure 1, in which a computational process as a whole is divided into an internal assemblage of program and data structures I will collectively call the **structural field**, coupled with an internal process that examines and manipulates these structures. In computer science this inner process (or 'homunculus') is typically called the *interpreter*; in order to avoid confusion with semantic notions of interpretation, I will call it the **processor**. While models of reflection for concurrent systems could undoubtedly be formulated, the claim I make here is only that the architecture I will describe is general for calculi of this serial (i.e., single processor) sort.

I will use the term '**structure**' for elements of the structural field, all of which are assumed to be inside the machine; the word will never be used for abstract mathematical or other "external" entities, such as numbers, functions, or radios.[5] Consequently, I call **metastructural** any structure that designates another structure, reserving **metasyntactic** for *expressions designating linguistic entities or expressions*.[6] Given an interest in internal self-reference, it is clear that both structural field and processor, as well as numbers and functions and the like, must be part of the semantic domain. Note also that the property of being *metastructural* is to be distinguished from the orthogonal property of being *higher-order*, in which terms and arguments may designate functions of any degree (2-Lisp and 3-Lisp will have both properties).[7]

[5] Although this terminology may be confusing for semanticists who think of a "structure" as a model, I want to avoid calling internal ingredients *expressions*, since the latter term connotes linguistic or notational entities. What I am aiming for is a concept covering both (i) what we would traditionally call data structures, and (ii) the "internal representation" of the program, which we can indirectly use to categorize what we would in ordinary English call the structure of the overall process or agent.

[6] Because of the constraints of appropriate causal connection, the metastructural capability must be provided by primitive quotation mechanisms, as opposed simply to being able to *model* or *designate* syntax—something virtually any calculus can do, using for example Gödel numbering.

[7] Most programming languages, such as Fortran and Algol 60, are neither higher-order nor metastructural; the ⊠-calculus is the former but not the latter, whereas Lisp 1.5 is the latter but not the former (dynamic scoping is a contextual protocol that, coupled with the meta-structural facilities, allows Lisp 1.5 partially to compensate for the fact that it is only first-order. At least some incarnations of Scheme, on the other hand, are both higher-order and metastructural (although Scheme's metastructural powers are expressly limited). As will emerge, 3-Lisp's combination of metastructural and higher-order properties are essential to its reflective capabilities.

3 A Framework for Computational Semantics

Given this background, turn first to questions of semantics. In the simplest case, semantics is taken to involve a mapping, possibly contextually relativized, from a syntactic to semantic domain, as shown in figure 2. The mapping ϕ is typically called an **interpretation function** (to be distinguished, as noted above, from the standard computer science notion of an "interpreter"). Interpretation functions are usually specified inductively, with respect to the compositional structure of the elements of the syntactic domain, which in turn is typically taken to be a set of entities of a syntactic or linguistic sort. Semantic domains may be of any type whatsoever, including domains of behaviour; in reflective systems they will typically include the syntactic and structural domains as proper parts. In this paper, to minimize confusion, I will use a variety of different meta-theoretic variables for different *kinds* of semantic relationship; in the general case, I will use the variable *s* and its cognates (s_1, s_2, s', etc.) to denote symbols or signs, and for any semantic value *d* will say that *s signifies d*, or conversely that *d* is the *significance* or *interpretation* of s.

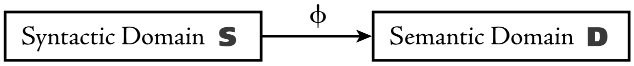


Figure 2 — Simple Semantic Interpretation

It is a fundamental tenet of the proposed approach to reflection to recognize that, in a computational setting, there are several different semantic relationships—not different ways of characterizing one and the same relationship (as operational and denotational semantical accounts are sometimes taken to be, for example), but *genuinely distinct relationships*. These different relationships make for a more complex semantic framework than is standard in logic and model theory, as do ambiguities in the use of words like 'program.' In many settings, such as in purely extensional functional programming languages, such distinctions are relatively inconsequential, and can be harmlessly glossed or elided. But in cases of reflection, self-reference, and metastructural processing, these distinctions, which in other circumstances may seem minor, play a much more important role.

Since the semantical theory adopted to *analyse* 3-Lisp will be at least partially embedded *within* 3-Lisp, choice of semantical framework affects the formal architecture and design. My approach, therefore, will be to start with basic and simple intuitions, and to identify a finer-grained set of distinctions than are usually employed. I will briefly consider the issue of how the contemporary practice of programming language semantics would be reconstructed in its terms, but the complexities involved in answering that question adequately would take us beyond the scope of the present paper.

Given these preliminaries, I will distinguish three things:

1. The *external* objects and events in the world in which a computational process is embedded—including both real-world objects such as cars and caviar, and set-theoretic abstractions such as numbers and functions (that is: I will adopt a kind of pan-Platonic idealism about mathematical entities);
1. The *internal* elements, structures, or processes inside the computer, including data structures, program representations, execution sequences and so forth (these are all formal objects, in the sense that computation is formal symbol manipulation[c]); and
2. *Notational* or *communicational expressions*, in some externally observable and consensually established medium of interaction, such as strings of characters, streams of words, or sequences of display images on a computer terminal.

The third set—of expressions—are assumed to include the constituents of communication with the computational process (by human agents or other computational processes); the middle set are the ingredients of the process with which those communicating external agents and processes interact; and the first (at least presumptively) are the elements of the world or "subject matter" about which that communication is held. In the human case, the three domains would correspond, respectively, to *world*, *mind*, and *language*.

[c] Even at the time this paper was published I was critical of the idea that computation could adequately be understood as formal symbol manipulation; I believe that the phrasing "in the sense that" was meant to signal (rather ineffectively) some distancing of my own view from that then-universal assumption. It was not until 1986 that I explicitly argued against such a construal. See «ref "From Symbols to Knowledge", and AOS.»

It is a theoretical truism that the third domain of objects—the elements of communication—are semantic, in the sense of being meaningful, serving as vehicles of meaning, carrying information, or some such. In this work I will take the middle set to be semantic as well—i.e. will assume that *internal structures* are bearers of meaning, information, and/or content. Distinguishing between the semantics of communicative expressions and the semantics of internal structures will be one of the main features of the framework I adopt. It should be noted, however, that in spite of my endorsing the reality of internal structures, and assuming the reality of the embedding world, it is nonetheless true that in the cases I will consider (i.e., ignoring sensors and manipulators), the only things that actually *happen* with computers are communicative interactions. For example, in a case that I might informally describe as "asking my Lisp machine what the square root of two is," what in fact happens, concretely, is that I type an expression such as `(SQRT 2.0)` at the computer, and receive back *some other expression*, probably quite like `1.414`, by way of response. What matters, for our purposes, is that the interaction is carried out entirely in terms of expressions; no structures, numbers, or functions are part of the interactional event (in particular, it is metaphysically precluded, given the presumed philosophy of mathematics, for a computer to *return the square root of two*). The denotation or participation or relevance of any of more abstract objects, such as numbers, must be inferred from, and mediated through, the communicative act.

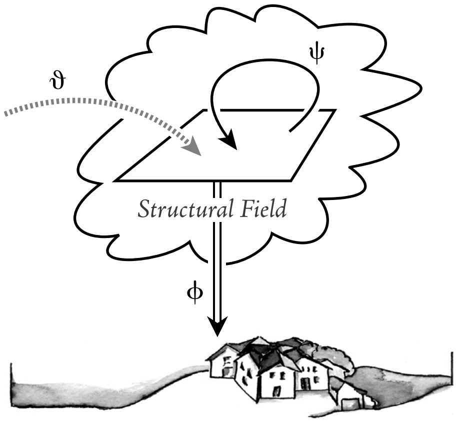


Figure 3 — Semantic Relationships in a Computational Process

I will begin to analyse this complex of relationships using the terminology suggested in figure 3. By ⊠, very simply, I will refer to the relationship between *external notational expressions* and *internal structures*; by '⊠' I will refer to the processes and behaviours those structural field elements engender (thus '⊠' is inherently temporal); and by '⊠' I will to the entities in the world that they designate. For mnemonic convenience, relations '⊠' and '⊠' have been named to suggest *philosophy* and *psychology*, respectively, since a study of '⊠' is a study of the relationship between structures and the world, whereas a study of '⊠' is a study of the relationships among symbols, all of which are "within the head" (of person or machine).

Since computation is inherently temporal, the semantic analysis must deal explicitly with relationships across the passage of time. In figure 4, therefore, I have unfolded the diagram of figure 3 across a unit of time, so as to get at a full configuration of these relationships. Entities n_1 and n_2 are intended to be linguistic or communicative entities, as described above;[8] s_1 and s_2 are internal structures over which internal processing is defined. The relationship ⊠, which I will call **internalisation** (and its inverse, ⊠$^{-1}$, **externalisation**) relates these two kinds of object, as is appropriate given the device or process in question (thus I will say, in addition, that n_1 **notates** s_1). For example, in first order logic n_1 and n_2 would be expressions, written with letters, spaces, '⊠' and '⊠' signs, etc.; to the extent that s_1 and s_2 could be said to exist, in logic, they would be something like abstract derivation tree types of the corresponding first-order formulae. In Lisp, as we will see, n_1 and n_2 would be the input and output expressions, written with letters and parentheses, or perhaps with boxes and arrows; s_1 and s_2 would be the corresponding cons cells in the s-expression heap.

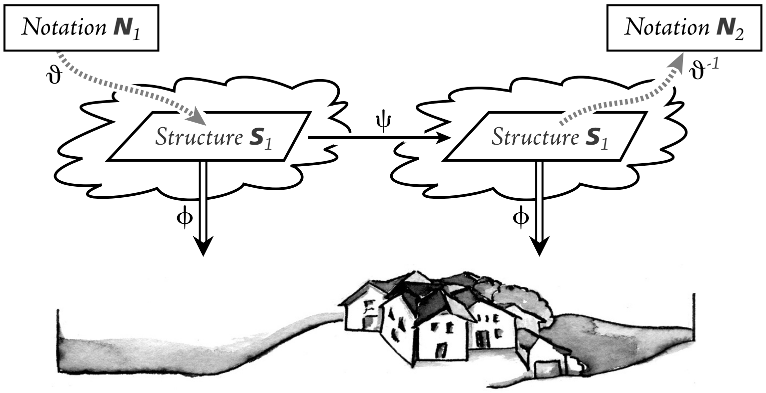


Figure 4 — A Framework/or Computational Semantics

[8] That is: the variable 'n_1' and its cognates are used in this text is as a meta-level variable to denote a linguistic or communication expression; etc.

In contrast, d_1 and d_2 are elements or fragments of the embedding world, and ⊠ is the relationship that internal structures bear to them. ⊠, in other words, is semantics' so-called "interpretation function" that makes explicit what I will call the **designation** of internal structures (not, note, the designation of linguistic expressions or terms, which would be described by ∉∘ ∉). The relationship between my mental token representing T. S. Eliot, for example, and the poet himself, would be formulated as part of ∉ whereas the relationship between the public name 'T. S. Eliot' and the poet would be expressed as ∉(∉("T. S. Eliot")) = T. S. Eliot. Similarly, ∉would relate an internal "numeral" structure (say, the numeral 3) to the corresponding number—if I can be permitted to use the word 'numeral' to refer to internal structures as well as to external expressions. As mentioned at the outset, my focus on ∉is evidence of the permeating semantical assumption that all structures have designations—or, to put it another way, that in the computational realm I am considering, all structures are taken to be symbols.[9]

In contrast to ∉and ∉ the relation ∉always (and necessarily, since it does not have access to anything else) relates some internal structures to others, or to behaviours over them. To the extent that it would make sense to talk of a ∉in logic, it would be approximately the formally computed derivability relationship (∉); in a natural deduction or resolution schemes, ∉would be a subset of the derivability relationship, picking out the particular inference procedures those regimens adopt. In a computational setting, however, ∉would be the function computed by the processor (i.e., in traditional Lisp it is *evaluation*).

[9] For what I might call *declarative languages*, there is a natural account of the relationship between linguistic expressions and in-the-world designations that need not make crucial reference to issues of processing (to which I will turn in a moment). It is for such languages, in particular, that the composition ∉∘ ∉(call it ∉), would be formulated. For obvious reasons, it is ∉ that is typically studied in mathematical model theory and logic, since those fields do not deal in any crucial way with the *active use* of the languages they study. In logic, for example, ∉ would be the interpretation function of standard model theory. In what I will call *computational* languages, on the other hand, questions of processing (∉) do arise for all aspects of significance—and so, in a vaguely Wittgensteinian sense, ∉ cannot in general be explicated independent of ∉

The relationships ⋹, ⋹, and ⋹have different relative importance in different semiotic disciplines, and relationships among them have been given different names. For example, ⋹is usually ignored in logic, and there is little tendency to view the study of ⋹, called proof theory, as *semantical*, although it is always related to semantics, as in proving soundness and completeness.[10] In addition, there are a variety of "independence" claims that have arisen in different fields. That ⋹does not uniquely determine ⋹, for example, is the "psychology narrowly construed" and concomitant methodological solipsism of Putnam, Fodor, and others.[11] That ⋹is usually specifiable compositionally and independently of ⋹or ⋹is essentially a statement of the autonomy thesis for language. Similarly, when ⋹cannot be specified independently of ⋹, computer science will say that a programming language "cannot be parsed except at runtime" (a property exemplified by Teco and the first versions of Smalltalk[12]).

A thorough analysis of these semantic relationships, however, and of the relationships among them, is the subject of a different paper. For present purposes I need not take a stand on which of ⋹, ⋹, or ⋹has a prior claim on being "semantics," but it will help to have some English terminology for some of these relations, in order not to have to devolve into formalism. For discussion, therefore, I will refer to the "⋹" of a structure as its **declarative import**, and to its "⋹" as its **procedural consequence**.[d] It is also convenient to identify some of the situations when two of the six entities (n_1, n_2, s_1, s_2, d_1 and d_2) are identical. In particular, I will say that s_1 is **self-referential** if

[10] Soundness and completeness can be expressed as ⋹(s_1,s_1) **ƛ** [⋹(s_1) # ⋹(d_1)], if one takes ⋹to be a relation, and ⋹to be an inverse satisfaction relationship between sentences and possible worlds that satisfy them.

[11] See Fodor (1980).

[12] Teco ("text editor and corrector") was a string-processing language which ran on the "Incompatible Time Sharing Systems" (ITS) at the MIT Artificial Intelligence Lab in the 1970s. It is now remembered primarily as the programming language in which the initial versions of the still-popular text editor EMACS were written. Smalltalk, an object-centered, dynamically-typed, "reflective" programming language, was developed at the Xerox Palo Alto Research Center (PARC) by Alan Kay and his colleagues, also during the 1970s.

[d] «This was already said. Check that—but also check all the terminology used for these relations; there is redundancy and confusion throughout.»

$s_1 = d_1$, that ∉**de-references** s_1 if $s_2 = d_1$, and that ∉is **designation-preserving** (at s_1) when $d_1 = d_2$ (as it always is, for example, in the ∉calculus, where at least in the standard model ∉—some combination of ∉and ∉reduction—does not alter the interpretation).

It is natural to ask what a *program* is, what *programming language semantics* gives an account of, and how (this is a related question) ∉ and ∉relate in the programming language case. An adequate answer to this, however, introduces a maze of complexity that I will have to defer to future work. To appreciate some of the difficulties, note that there are two different ways in which we can conceive of a program, suggesting different semantical analyses.[e] On the one hand, a program can be viewed as a linguistic object that *describes* or *signifies* a computational process consisting of the data structures and activities that result from (or arise during) its execution. In this sense a program is primarily a *referential* or *communicative* entity—not so much playing a causal role within a computational process so much as existing outside the process and representing it. Putting aside for a moment the question of whom it is meant to communicate to, I would simply that on such a reading a program is in the domain of ∉, and, roughly, that ∉∘ ∉of such an expression would be the computation described. The same characterization would, of course, apply to a specification; indeed, the only salient difference might be that a specification would avoid using non-effective concepts in describing behaviour. One would expect specifications to be stated in a declarative language (in the sense defined in footnote ■■), since specifications are not, per se, intended to be executed or run, even though they speak about behaviours or computations. Thus, for program or specification b describing computational process c, we would have (for the relevant language) something like ∉(∉(b))=c. If b were a *program*, there would be an additional constraint that the program somehow play a causal role in engendering the computational process c that it is taken to describe.

There is an alternative conception, however, which places the

[e] «This may be the first occurrence of my on-going attention to the differences between and among *specificational*, *ingrediential*, and *communicational* views of programs. Refer back to the 2010 perspective at the outset; and forward to the places where I have the pictures, etc.»

program *inside* the machine, as a causal participant in the behaviour that results. This view is closer to the one implicitly adopted in figure 1, and I believe that it is closer to the way in which a Lisp program *must be semantically analysed if we are to understand Lisp's emergent reflective properties*. In some ways this different view has a von Neumann character, in the sense of equating program and data. On this view, the more appropriate equation would seem to be ǂ(ǂ(*b*))=*c*, since one would expect the processing of the program to yield the appropriate behaviour. One would seem to have to reconcile this equation with that in the previous paragraph, although it is not clear that this would be possible.[f]

Disentangling these points will require further work. What I can say here is that programming language semantics seems to focus on what, in the terminology I am using, would seem be an amalgam of ǂand ǂ For our purposes I need only note that we will have to keep ǂand ǂ*strictly separate*, while recognising (because of context relativity and non-local effects) that just because they are distinct does not mean they are independent. Formally, I would need to specify a general significance function ǂ[13] which recursively specifies ǂand ǂtogether. In particular, given any structure s_1, and any state of the processor and the rest of the field (encoded, say, in an environment, continuation, and perhaps a store), ǂwill specify the structure, configuration, and state that would result (i.e., it will specify the *use* of s_1), and also the signifying relationship that s_1 bears to the world. For example, given a Lisp structure of the form `(+ 1 (PROG (SETQ A 2) A))`, ǂ would specify that the whole structure designated the number three, that it would "return" (i.e., that its procedural consequence would be) the numeral `3`, and that the machine would be left in a state in which the binding of the variable `A` was changed to the (structural)

[13] This is what was done in «ref TR».

[f] «I believe this last sentence is either confused or wrong. Think about it and fix as appropriate.»

[g] Computer science talks about a variable being "bound to" something—namely, to its *value*—though, as evident in the semantical reconstruction being carried out here, that usually means to a co-referential structure. Strictly speaking, that is, a programming language variable would be bound to a *numeral*, not to a *number*—and should be so described, in contexts in which the differences between numerals and numbers are significant. In mathemat-

numeral 2.[9]

Before leaving semantics completely, it is instructive to apply these various distinctions to traditional Lisp. I said above that all interaction with computational processes is mediated by communication; this can be stated in the present terminology by noting that $\notin$and $\notin^{\mathrm{I}}$ (internalization and externalization) are a part of any interaction. Thus Lisp's "READ-EVAL-PRINT" loop is mirrored in this analysis as an iterated version of $\notin^{\mathrm{I}} \circ \notin \circ \notin$(i.e., if n_{I} is an expression that you type as input to a Lisp system, returning n_2 as output, then $n_2 = \notin^{\mathrm{I}}(\notin(\notin(n_{\mathrm{I}})))$. The Lisp structural field, as it happens, has an extremely simple compositional structure, based on a binary directed graph of atomic elements called *cons-cells*, extended with atoms, numerals, and so forth. The linguistic or communicative expressions that we use to represent Lisp programs—the formal language objects that we edit with our editors and print in books and on terminal screens—is a separate lexical (or sometimes graphical) entity with its own syntax (parentheses and identifiers in the lexical case; boxes and arrows in the graphical).

In Lisp there is a relatively close correspondence between expressions and structures; it is one-to-one in the graphical case, but the standard lexical notation is both ambiguous (because of shared tails) and incomplete (because of its inability to represent cyclical structures). The correspondence need not have been as close as it is; the process of converting from external syntax or notation to internal structure could involve arbitrary amounts of computation, as evidenced by read macros and other syntactic or notational devices. But the important point is that it is *structural field elements*, not *notations*, over which most Lisp operations are defined. If you type "(RPLACA '(A . B) 'C)", for example, the processor will (as expected)

ics and logic, variables are likely, if bound to anything, to be bound to *numbers*—i.e., to what is here being called declarative import. Moreover, it is also more common in logic and mathematics to describe a variable as "bound *by*" something—namely, bound by *quantifiers*, *scoping constructs*, etc. This is just one small instance of the general phenomenon of computer science's using, as technical terminology, vocabulary and phrasings derived from logic, but in its own distinct ways. Sometimes, as here, the differences are subtle, and not usually distracting; sometimes, as with the word 'semantics,' they are major, and cause of considerable confusion. See AOS.

first create and then change the CAR (first element) of a field structure; it will not back up your terminal and erase the eleventh character of the *expression* that you typed as input (if that were even physically possible). Similarly, Lisp atoms are field elements, not to be confused with their lexical representations (sometimes called 'P-names' or "print-names"). Again, quoted forms such as (QUOTE ABC) designate structural field elements, not input strings. The form (QUOTE ___), in other words, is a *structural* quotation operator; *notational* quotation is different, usually notated with string quotes (as in "ABC").[14]

4 Evaluation Considered Harmful[h]

The claim that all three relationships (ȼ, ȼ, and ɇ) figure crucially in an account of Lisp is not a formal one. It makes an empirical claim on the minds of programmers, and cannot be settled by pointing to any current theories or implementations. Arguments in its behalf would point to the fact that Lisp's numerals are universally taken to designate numbers, and that the atoms T and NIL (at least in predicative contexts) are similarly understood to stand for truth and falsity—no one could learn Lisp without learning these facts, and the behaviour of Lisp systems is only intelligible on such an

[14] The string "(QUOTE ABC)" notates a structure that designates another structure that in turn could be notated with the string "ABC". The string " "ABC" ", on the other hand, notates a structure that designates the string "ABC" directly.

[h] This section title is a play on Edsger W. Dijkstra's legendary "GO TO Statement Considered Harmful" (*Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147–48). No computer scientist in the 1980s would have failed to recognize the illusion; the *Communications of the ACM* (Association for Computing Machinery) was the première professional computer science journal at the time, and Dijkstra's letter was widely taken to have inaugurated serious theoretical analysis of programming. Cf. this note from the History of Computing Project:

> "In 1968 Edsger Dijkstra laid the foundation stone in the march towards creating structure in the domain of programming by writing, not a scholarly paper on the subject, but instead a letter to the editor entitled "GO TO Statement Considered Harmful". (*Comm. ACM*, August 1968) The movement to develop reliable software was underway."

See http://www.thocp.net/biographies/dijkstra_edsger.htm

[i] «Put in a pointer to (and discussion of) the "normatively governed effective mechanism" construal of logic and other intentional systems in other papers.»

assumption.[i] In what follows I will therefore state, without qualification, that '3' (i.e., the structural numeral notated by the string character "3") designates three; that `T` designates truth, that `(EQ 'A 'B)` designates falsity, etc. In a similar spirit, I will claim that the structure `(CAR '(A . B))` *designates* the atom `A`; this is manifested by the fact that people, in describing Lisp, use expressions such as "If the `CAR` of the list is `LAMBDA`, then it is a procedure," where the ingredient term "the `CAR` of the list" is used as an *English referring expression*—specifically as a *singular term*—not as a quoted fragment of Lisp (and English, or natural language generally, is by definition the locus of what designation is). `(QUOTE A)`, or `'A`, is another way of *naming* or *designating* the atom `A`; that is just what quotation is. By the same token, I will take such atoms as `CAR` and `+` to name or designate the obvious corresponding functions.

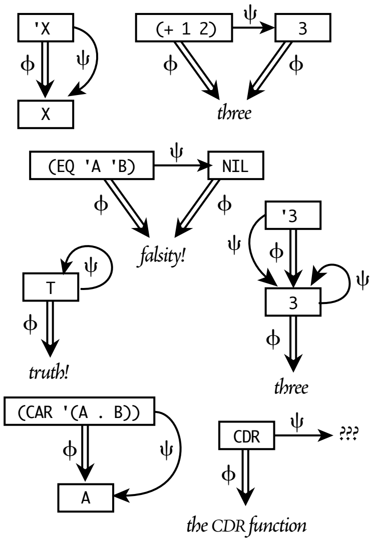


Figure 5 — Lisp Evaluation vs. Designation: Some Examples

What, then, is the relationship between the declarative import (Φ) of Lisp structures and their procedural consequence (Ψ)? Inspection of the superficially rather bewildering data given in figure 5 shows that Lisp obeys the following constraint, where *S* is the domain of structural field elements (more must be said about Ψ in those cases where $\Phi(\Psi(s)) = \Phi(s)$, since the identity function would satisfy this equation):

$$\forall s \in S \text{ if } \Phi(s) \in S \text{ then } \Psi(s) = \Phi(s) \qquad [1]$$
$$\text{else } \Phi(\Psi(s)) = \Phi(s)$$

All Lisps, including Scheme,[15] in other words, *de-reference* any structure whose designation is another structure, but will return a *co-designating* structure for any whose designation is external to the machine. This regularity, which generates the variety of cases illustrated in figure 5, is depicted in figure 6. Whereas evaluation is often

[15] Steele and Sussman (1978a).

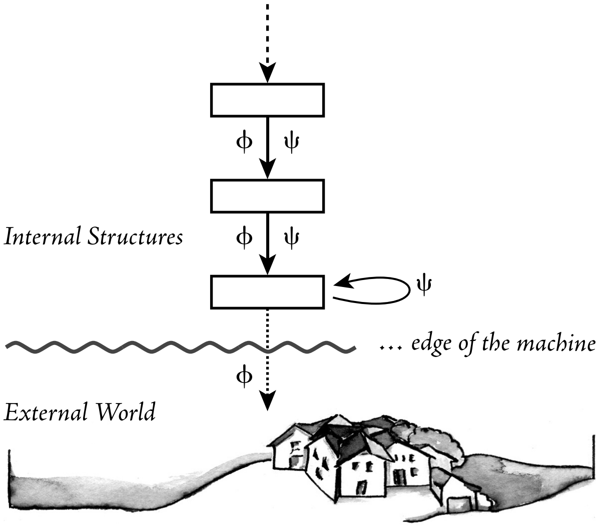


Figure 6 — Lisp's "De-reference if You Can" Evaluation Protocol

thought to correspond to the semantic interpretation function ϕ—in other words, and therefore to have type *expressions* → *values*, evaluation in Lisp is often a *designation-preserving* operation. In fact, it is a metaphysical fact that no computer can evaluate a structure such as `(+ 2 3)`, if that means "returning what is designated," at least on the Platonist understanding of number I am working with, any more than it can evaluate the name *Hesperus*, or than it is likely to be able to evaluate the name *peanut butter*.

I take it as self-evident that obeying equation [1] is anomalous. It implies, among other things, that even if in a case in which one knows what *y* is, and knows that *x* evaluates to *y*, one still does not know what *x* designates. It also licenses such semantic anomalies as `(+ 1 '2)`, which—contrary, I would argue, both to common and to theoretical sense—will evaluate to (the structure!) `3` in all extant Lisps. Informally, I will say that Lisp's evaluator *crosses semantical levels*, and therefore obscures the difference between simplification and designation. Given that processors cannot always de-reference (since by assumption the co-domain is limited to the structural field), the only semantically consistent non-level-crossing behaviour they can exhibit in general is to preserve designation. It seems, therefore, that they should always *simplify*, and therefore obey the following constraint (diagrammed in figure 7):

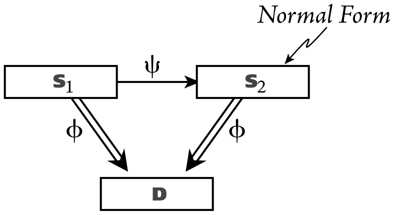


Figure 7 — A Normalisation Protocol

ǂs ǂ*S* ǂ(ǂ(*s*)) = ǂ(*s*) ǂnormal-form(ǂ(*s*)) [2]

The content of this equation clearly depends entirely on the content of the predicate "normal-form" (if "normal-form" were l*x*.true, then ǂ could be the identity function). In the ǂcalculus, the notion of normal-formedness is defined in terms of the processing protocols (ǂ and ǂreduction), but I cannot use any such definition here, on threat of circularity. Instead, I will say that a structure is in normal form if and only if it satisfies the following three independent conditions:

1. It is **context-independent**, in the sense of having the same declarative (ǂ) and procedural (ǂ) import independent of the context of use;
2. It is **side-effect-free**, implying that the processing of the structure will have no effect on the structural field, processor state, or external world; and
3. It is **stable**, meaning that it must simplify to itself in all contexts, so that ǂwill be idempotent.

We would then have to prove, given a language specification, that equation [2] is satisfied (as it is in the case of 2-Lisp and 3-Lisp)

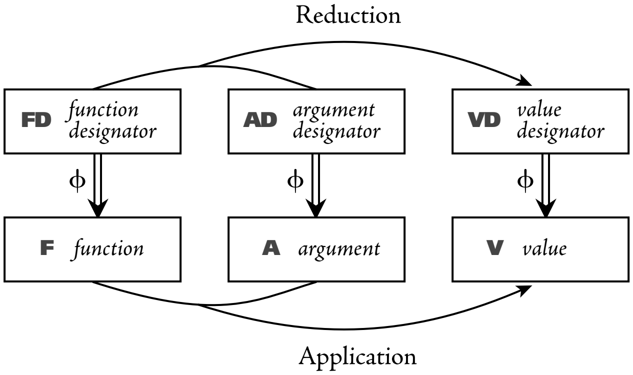


Figure 8 —Application vs. Reduction

Two notes. First, I will not use the terms 'evaluate' or 'value' for expressions or structures, referring instead to **normalisation** for ǂ and **designation** for ǂ I will sometimes call the result of normalising a structure its *result* or what it *returns*. There is also a problem with the terms 'apply' and 'application.' In standard Lisps, APPLY is (the name of) a function from structures and arguments onto values, but like 'evaluate', its use is rife with use/

mention confusions. As illustrated in figure 8, I will use 'apply' for mathematical function application—i.e., to refer to a relationship between a function, some arguments, and the value of the function applied to those arguments—and the term '**reduce**' to relate the three structures that *designate* functions, arguments, and values, respectively. Note that this terminological practice retains use of the term 'value' (as, for example, in the previous sentence), but only to name that entity onto which a mathematical function maps its arguments.

Second, the idea of a normalising processor depends on the idea that symbolic structures have a semantic significance *prior to, and independent of*, the way in which they are treated by the processor.[j] Without this assumption we could not even *ask* about the semantic character of the Lisp (or any other) processor, let alone suggest a cleaner version. Without such an assumption, more generally, one cannot say that a given processor is correct, or coherent, or incoherent; it would merely be what it is. Given one account of what it did (such as an implementation), one could compare that to another account (such as a specification). One could also prove that it had certain properties, such as that it always terminated, or that it used resources in certain ways. One could even prove properties of programs written in the language it runs (from a specification of the ALGOL processor, for example, one might prove that a particular program sorted its input). However, none of these questions deal with the question I am taking to be more fundamental: about the semantical nature of the processor itself. I am not satisfied to say that the semantics of `(CAR '(A . B))` is `A` *because that is how the processor is defined*; rather, I want to say that the processor was defined that way *because* `A` *is what* `(CAR I (A . B))` *designates*. Semantics, in other words, should be a tool with which to *judge* systems, not merely a method of *describing* them.

5 2·Lisp: A Semantically Rationalised Dialect

Having torn apart the notion of evaluation into two constituent notions (designation and simplification), we need to start at the be-

[j] «Talk about this in relation to Amala, Mike Dixon's thesis, errors in the definition of factorial, etc.—and to subsequent semantical inquiry (also to logic).»

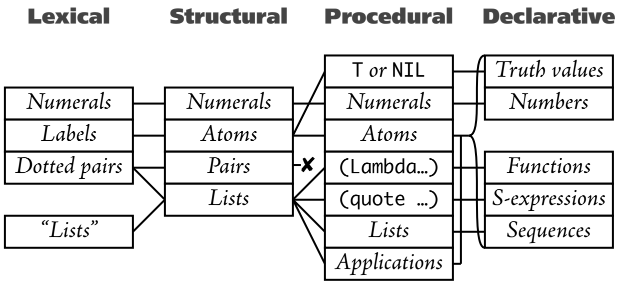


Figure 9 — The Category Structure of Lisp 1.5

ginning, and build Lisp over again. What I am calling **2-Lisp** is a proposed result. Some summary comments can be made.

First, I have reconstructed what I call the **category structure** of Lisp, requiring that the categories into which Lisp structures are sorted, for various purposes, "line up" (giving the dialect a property I will call **category alignment**). More specifically, Lisp expressions are sorted into categories by *notation*, *structure* (atoms, cons pairs, numerals), *procedural treatment* (the "dispatch" inside the traditional EVAL), and *declarative semantics* (the type of object designated). As illustrated in figure 9, these categories are traditionally not aligned; *lists*, a derived structure type, include some of the pairs and one atom (NIL); the procedural regimen (⋹) treats some pairs (those with LAMBDA in the CAR) in one way, most atoms (except T and NIL) in another, and so forth. In 2-Lisp, in contrast, I have required the notational, structural, procedural, and semantic categories to correspond, as much as practicable, one-to-one, as illustrated in figure 10 (this is a bit of an oversimplification, since atoms and pairs—representing

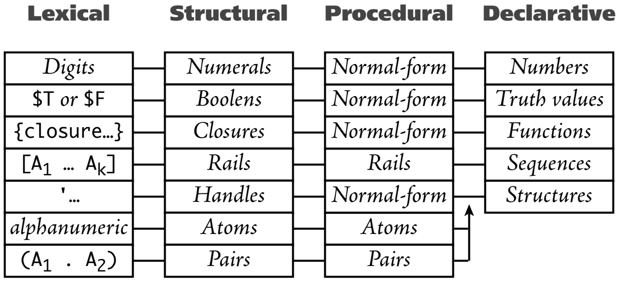


Figure 10 — The Category Structure of 2-Lisp and 3-Lisp

arbitrary variables and arbitrary function application structures or redexes—can designate entities of any semantic type).

2-Lisp is summarized in the sidebar ("An Overview of 2-Lisp," starting below); some additional comments can be made here. Like most mathematical and logical languages, 2-Lisp is almost entirely "**declaratively extensional**". Thus `(+ 1 2)`, an abbreviation for `(+ . [1 2])`, designates the value of the application of the function designated by the atom `+` to the sequence of numbers designated by the rail `[1 2]`. In other words, `(+ 1 2)` designates the number three, of which the numeral `3` is the normal-form designator; `(+ 1 2)` therefore normalises to the numeral `3`, as expected. 2-Lisp is also usually call-by-value (what we might call "**procedurally extensional**"), in the sense that procedures by and large normalise their arguments. Thus the structure `(+ 1 (BLOCK (PRINT "HELLO") 2)` will normalise to `3`, printing out `"HELLO"` in the process.

Many properties of Lisp that must normally be posited in an ad

An Overview of 2-Lisp

Begin with objects. Ignoring input/output categories such as *characters*, *strings*, and *streams*, there are seven 2-Lisp structure types, as illustrated in Table 1. The numerals (notated as usual) and the two Boolean constants (notated '`$T`' and '`$F`') are unique (i.e., canonical), atomic, normal-form designators of numbers and truth-values, respectively. Rails (notated '[A$_1$ A$_2$ … A$_k$]') designate sequences; they resemble standard Lisp lists, but are distinguished from pairs in order to avoid category confusion, and are given their own name in order to avoid confusion with *sequences*, *vectors*, and *tuples*, all of which are normally taken to be Platonic ideals.

All atoms are used as variables (i.e., as context-dependent names); as a consequence, no atom is normal-form, and no atom will ever be returned as the result of processing a structure (although a designator of an atom may be). Pairs (sometimes also called redexes—notated '(A$_1$. A$_2$)'—designate the value of the function designated by their `CAR` (i.e., A$_1$) applied to the arguments designated by their `CDR` (A$_2$). By taking notational form '(A$_1$ A$_2$ … A$_k$)' to abbreviate '(A$_1$. [A$_2$ … A$_k$])' instead of Lisp's traditional '(A$_1$. (A$_2$. … (A$_k$. NIL)…)))', we preserve the standard look of Lisp programs, without sacrificing category alignment. (Note that 2-Lisp has no distinguished atom `NIL`, and '`()`' is a notational error—corresponding to no structural field element.) Closures (notated hoc way fall out directly from this analysis. For example, it normally requires explicit statement that some atoms, such as T and NIL and all numerals, are self-evaluating; in 2-Lisp, the fact that the Boolean constants are self-normalising follows directly from the fact that they are normal-form designators. Similarly, closures are a natural category, and distinguishable from the functions they designate (there is ambiguity, in Scheme, as to whether the value of + is a function or a closure). Finally, because of category alignment, if X designates a sequence of the first three numbers (i.e., it is bound to the rail [1 2 3]), then (+ . X) will designate the number five and normalise to the numeral 5; no metatheoretic machinery is needed for this "un-currying" operation (in regular Lisps one must use (APPLY '+ X); in Scheme, (APPLY + X)).

Numerous properties of 2-Lisp will be ignored in this paper. The dialect is defined in Smith (1982) to include side-effects, intensional procedures (procedures which do not normalise their arguments),

An Overview of 2-Lisp (cont'd)

'{CLOSURE: … }') are normal-form function designators, but they are not canonical, since it is not in general decidable whether two structures designate the same function. Finally, handles are unique normal-form designators of all structures; they are notated with a leading single quote mark (thus 'A notates the handle of the atom notated A, and '(A . B) notates the handle of the pair notated (A . B), etc. Because designation and simplification are orthogonal, quotation is a structural primitive, not a special procedure (although QUOTE is easy to define as a user function in 3-Lisp).

Turn next to the functions (and use '⇒' to mean 'normalises to'). There are the usual arithmetic primitives (+, -, *, and /). Identity (signified with '=') is computable over the full semantic domain except functions; thus (= 3 (+ 1 2)) ⇒ $T, but (= + (LAMBDA [X] (+ X X))) will generate a processing error, even though it designates truth. The traditionally rather atheoretical difference between EQ and EQUAL turns out to be an expected difference in granularity between the identity of mathematical sequences and their syntactic designators; thus:†

```
(=  [1 2 3J  [1 2 3])  ⇒ $T
(= '[1 2 3] '[1 2 3])  ⇒ $F
(=  [1 2 3J '[1 2 3])  ⇒ $F
```

1ST and REST are the CAR/CDR analogues on both sequences and rails (i.e., and a variety of other sometimes-shunned properties, in part to show that this semantic reconstruction being argued for here is compatible with the full gamut of features found in real programming languages. Recursion is defined with respect to an analysis using explicit fixed-point operators. 2-Lisp is an eminently usable dialect (it subsumes Scheme but is more powerful, in part because of the metastructural access to closures), although it is ruthlessly semantically strict.

6 Self-Reference in 2·Lisp

Turn now to matters of self-reference.

Traditional Lisps provide names (EVAL and APPLY) for the primitive processor procedures; the 2-Lisp analogues are NORMALISE and REDUCE. Ignoring for a moment context arguments such as environments. and continuations, (NORMALISE '(+ 2 3)) designates the normal-form structure to which (+ 2 3) normalises, and therefore returns the handle

Type	**Numerals**	**Booleans**	**Handles**	**Closures**	**Rails**	**Atoms**	**Pairs**
Designation	Numbers	Truth values	Structures	Functions	Sequences	(ϕ of bndg)	(value of app)
Normal	Yes				Some	No	
Canonical	Yes			No		N/A	
Constructor	N/A			CCONS	RCONS	ACONS	PCONS
Notation	Digits	$T or $F	'*structure*	{closure …}	[s_1 … s_2]	Alphanumerics	(s_1 . s_2)

Table 1 — The 2-Lisp (and 3-Lisp) categories

have overloaded definitions); thus (1ST [10 20 30]) ⇒ 10; and (REST [10 20 30]) ⇒ [20 30]. CAR and CDR are defined over pairs; thus (CAR '(A . B)) ⇒ 'A (because it designates A), and (CDR '(+ 1 2)) ⇒ '[1 2]. The pair constructor is called PCONS (thus (PCONS 'A 'B) ⇒ '(A . B); the corresponding constructors for atoms, rails, and closures are ACONS, RCONS, and CCONS, respectively. There are eleven primitive characteristic predicates—seven for the internal structural types (ATOM, PAIR, RAIL, BOOLEAN, NUMERAL, CLOSURE, and HANDLE) and four for the external types (NUMBER, TRUTH-VALUE, SEQUENCE, and FUNCTION). Thus:

'5. Similarly:

```
(NORMALISE '(CAR '(A. B)))     ⇒ ''A
(NORMALISE (PCONS '= '[2 3J))  ⇒ '$F
(REDUCE '1ST '[10 20 30J)      ⇒ '10
```

More generally—and entirely intuitively—the basic idea is that ∉(NORMALISE) = ∉, to be contrasted with ∉(↓), which is approximately ∉, except that because ↓ is a partial function we have ∉(↓ ∘ NORMALISE) = ∉. Given these equations, the behaviour illustrated in the foregoing examples is forced by general semantical considerations.

In any computational formalism able to model its own syntax and structures,[16] it is possible to construct what are commonly known

[16] Virtually any language has the requisite power to do this kind of modelling. In a language with metastructural abilities, the metacircular processor can represent programs for the MCP as themselves—this is always done

An Overview of 2-Lisp (cont'd)

```
(NUMBER 3)     ⇒ $T
(NUMERAL '3)   ⇒ $T
(NUMBER '3)    ⇒ $F
(FUNCTION +)   ⇒ $T
(FUNCTION '+)  ⇒ $F
```

Procedurally intensional IF and COND are defined as usual; BLOCK (as in Scheme) is like standard Lisp's PROGN. BODY, PATTERN, and ENVIRONMENT are the three selector functions on closures. Finally, functions are usually "defined" (i.e., conveniently designated in a contextually relative way) with structures of the form (LAMBDA SIMPLE ARGS BODY) (the term SIMPLE will be explained presently); thus (LAMBDA SIMPLE [X] (+ X X)) normalises to a closure that designates a function that doubles numbers:

```
((LAMBDA SIMPLE [X] (+ X X)) 4)   ⇒ 8
```

2-Lisp is *higher-order*, and therefore lexically scoped, like the ∉calculus and Scheme. As mentioned earlier, however, and illustrated with the handles in the previous paragraph, it is also *metastructural*, providing an explicit ability to name internal structures. Two primitive procedures, called UP and DOWN (usually abbreviated '↑' and '↓', respectively) help to mediate this metastrucas *metacircular interpreters*, which I will call **metacircular processors** (or MCPs)—"meta" because they operate on (and therefore terms within them designate) other formal structures, and "circular" because they do not constitute a definition of the processor in a prior, independently-understood language—but rather "define" the processor only in terms of itself. This circularity takes two forms. First, on the procedural side, MCPs must be *run* by the processor in order to yield any sort of behaviour (strictly speaking, that is, MCPs

in Lisp MCPs—but we need not define that to be an essential property. The term 'metacircular processor' is by no means strictly defined; there are various constraints that one might or might not put on it. My general approach has been to view as metacircular any *non-causally connected* model of a calculus within itself; thus the 3-Lisp reflective processor is *not* meta-circular, by my lights, because it *does* have the requisite causal connections, and is therefore an essential (not additional) part of the 3-Lisp architecture.

tural hierarchy (there is otherwise no way to add or remove quotes—'2 will normalise to '2 forever, never to 2. Specifically, ↑STRUC designates the normal-form designator of the designation of STRUC; i.e., ↑STRUC designates what STRUC normalises to (therefore ↑(+ 2 3) ⇒ '5). Thus (note that '↑' is call-by-value but not declaratively extensional):

```
 (LAMBDA SIMPLE [X] X)      — designates a function
'(LAMBDA SIMPLE [X] X)      — designates a pair or redex
↑(LAMBDA SIMPLE [X] X)      — designates a closure
```

Similarly, ↓STRUC designates the designation of the designation of STRUC, providing that the designation of STRUC is in normal-form (therefore ↓'2 ⇒ 2). ↓↑STRUC is always equivalent to STRUC, in terms of both designation and result; so is ↑↓STRUC when it is defined. Thus if DOUBLE is bound to (the result of normalising) (LAMBDA [X] (+ X X)), then (BODY DOUBLE) generates an error, since BODY is extensional and DOUBLE designates a function, but (BODY ↑DOUBLE) will designate the pair (+ X X).

†In the last case one structure designates a sequence and one a rail.

are *programs*, not *processors*). Second, the behaviour they would thereby engender—which is to say, the behaviour they must also therefore designate—can be discerned from them only if one knows beforehand what that behaviour is (i.e., what the processor does).[17] Nonetheless, such processors are pedagogically illuminating, and play a critical role in the development of procedural reflection.

The role of MCPs is illustrated in figure II, showing how, if we ever replace P in figure I with a process that results from P processing the metacircular processor MCP, it would still correctly engender the behaviour of any overall program. Taking processes to be functions from structures onto behaviour, therefore (whatever behaviour is—functions from initial to final states, say), and calling the primitive processor P, we should be able to prove that P(MCP) ≈ P, where by '≈' we mean *behaviourally equivalent in some appropriate sense*. The equivalence, of course, is in a certain sense global, or at the level of types; by and large the primitive processor and the processor resulting from the explicit running of the MCP cannot be arbitrarily mixed. If a variable is bound by the underlying processor P, it will not be able to be looked up by the metacircular code, for example. Similarly, if the metacircular processor encounters a control-structure primitive, such as a `THROW` or a `QUIT`, it will not cause the metacircular processor itself to exit prematurely, or to terminate. Rather, the point is that if an entire computation is run by the process that results from the explicit processing of the MCP by P, the results will be the same (modulo time) as if that entire computation had been carried out by P directly. MCPs, to put this in language to be used in providing

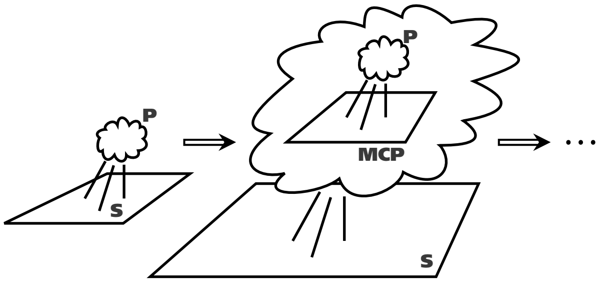


Figure II — Meta-Circular Processors

[17] Standard fixed point techniques are of no help in discharging these kinds of circularity, since what is at issue here is essentially self-*mention*, whereas although that terminology is commonly applied to recursive definitions, it would be more accurate to characterise recursion in terms of self-*use*.

genuine reflection, are not *causally connected* with the systems they model.

The reason that we cannot mix code for the underlying processor and code for the MCP and the reason that we ignored context arguments in the definitions above both have to do with the state of the processor P. In very simple systems (unordered rewrite rule systems, for example, and hardware architectures that put even the program counter into a memory location), the processor has no internal state, in the sense that it is in an identical configuration at every "click point" during the running of a program (i.e., all information is recorded explicitly in the structural field). But in more complex circumstances, there is always a certain amount of state to the processor that affects its behaviour with respect to any particular embedded fragment of code. In writing an MCP one must demonstrate, more or less explicitly, how the processor state affects the processing of object-level structures. By "more or less explicitly" I mean that the designer of the MCP has options: the state can be represented in explicit structures that are passed around as arguments within the processor, or it can be "absorbed" into the state of the processor running the MCP.[18]

The state of a processor for a recursively embedded functional language, of which Lisp is an example, is typically represented in an environment and a continuation, both in MCPs and in the standard metatheoretic accounts. (Note that these are notions that arise in the theory of Lisp, not in Lisp itself; except in self-referential or self-modelling dialects, user programs do not traffic in such entitles.) Most MCPs make the environment explicit. The control part of the state, however, encoded in a continuation, must also be made explicit in order to explain non-standard control operations, but in many MCPs (such as that in McCarthy (1965) and in Steele and Sussman's

[18] I say that a property or feature of an object language is **absorbed** in a metalanguage or theory just in case the metatheory uses the very same property to explain or describe the property of the object language. Thus conjunction is absorbed in standard model theories of first-order logics, because the semantics of P ∉Q is explained simply by conjoining the explanation of P and Q—specifically, in such a formula as "'P ∉Q' is true just in case 'P' is true *and* 'Q' is true".

«Add a note pointing to "The Correspondence Continuum"»

MCP for Scheme[19]) control context is absorbed.

Two versions of the 2-Lisp metacircular processor, one absorbing and one making explicit the continuation structure, are presented in sidebars on the following pages. Note that in both cases the underlying agency or anima is not reified; the "activity itself" remains entirely absorbed by the processor of the MCP. Nothing I have yet said (or in this paper will say) provides us with either name or mechanism to designate *process itself* (as opposed to structures and functional

[19] See for example Sussman and Steele (1978b).

Non-Continuation-Passing 2-LISP Metacircular Processor

```
(define READ-NORMALISE-PRINT
     (lambda simple [env stream]
         (block (prompt&rep1y(normalise (prompt&read stream) env)
                                 stream)
                 (read-normalise-print env stream))))

(define NORMALISE
     (lambda simple [struc env]
         (cond  [(normal struc) struc]
                [(atom struc) (binding struc env)]
                [(rail struc) (normalise-rail struc env)]
                [(pair struc) (reduce (car struc) (cdr struc) env)])))

(define REDUCE
     (lambda simple [proc args env]
         (let [[proc! (normalise proc env)]]
            (selectq (procedure-type proc!)
                [simple (let [[args! (normalise args env)]]
                              (if (primitive proc!)
                                  (reduce-primitive-simple proc! args!)
                                  (expand-closure proc! args!)))]
                [intensional (if (primitive proc!)
                                     (reduce-primitive-intensional proc! ▯args env)
                                     (expand-closure proc! ▯args))]
                [macro (normalise ▯(expand-closure proc! ▯args) env))]))))

(define NORMALISE-RAIL
     (lambda simple [rail env]
         (if (empty rail)
             (rcons)
             (prep (normalise (1st rail) env)
                    (normalise-rail (rest rail) env)))))

(define EXPAND-CLOSURE
     (lambda simple [proc! args!]
         (normalise (body proc!)
                     (bind (pattern proc!) args! (environment proc!)))))
```

behaviour over structure), and no method of obtaining causal access to an independent locus of active agency has been (or will be) provided.[20]

[20] The reason being that, as computer scientists, we as yet have no real theory of what processes are.
«Add a comment on this lack—and foreshadow work to come?»

Continuation-Passing 2-LISP Metacircular Processor

```
(define READ-NORMALISE-PRINT
     (lambda simple [env stream]
        (normalise (prompt&read stream) env
           (lambda simple [result]
              (block (prompt&reply result stream)
                 (read-normalise-print env stream))))))

(define NORMALISE
     (lambda simple [struc env cant]
        (cond [(normal struc) (cont struc)]
           [(atom struc) (cont (binding struc env»]
           [(rail struc) (normalise-rail struc env cant)]
           [(pair struc) (reduce (car struc) (cdr struc) env cant)])))

(define REDUCE
     (lambda simple [proc args env cant]
        (normalise proc env
           (lambda simple [proc!]
              (selectq (procedure-type proc!)
                 [simple (normalise args env
                              (lambda simple [args!]
                                 (if (primitive proc!)
                                     (reduce-primitive-simple proc! args! cont)
                                     (expand-closure proc! args! cont))]
                 [intensional (if (primitive proc!)
                                    (reduce-primitive-int proc! ▯args env cont)
                                    (expand-closure proc! ▯args cant))]
                 [macro (expand-closure proc! ▯args
                    (lambda simple [result]
                       (normalise ▯result env cant)))])))))

(define NORMALISE-RAIL
     (lambda simple [rail env cant]
        (if (empty rail)
            (cant (rcons))
            (normalise (1st rail) env
               (lambda simple [first!]
                  (normalise-rail (rest rail) env
                     (lambda simple [rest!]
                        (cant (prep first! rest!)))))))))

(define EXPAND-CLOSURE
     (lambda simple [proc! args! cant]
        (normalise(body proc!)
                    (bind (pattern proc!) args! (environment proc!))
                    cant)))
```

7 Procedural Reflection and 3·Lisp

Given the metacircular processors defined above, 3-Lisp can be non-effectively defined in a series of steps.

First, imagine a dialect of 2-Lisp, called 2-Lisp$_1$, where user programs are not run directly by the primitive processor, but by that processor running a copy of an MCP. Next, imagine 2-Lisp$_2$, in which the MCP in turn is not run by the primitive processor, but instead by the primitive processor running another copy of the MCP. And so on and so forth. 3-Lisp is essentially 2-Lisp$_\infty$, except that the MCP is changed in a critical way in order to provide the proper connection between levels. 3-Lisp, in other words, is what I will call a **reflective tower**, defined as equivalent to an infinite number of copies of an MCP-like program, run at the "top" by an (infinitely fleet) processor. The claim that 3-Lisp is well-founded is the claim that the limit exists—that is, that both sides of the following equation are sound:

$$\text{3-Lisp} \approx \lim_{n \to \infty}\left(\text{2-Lisp}_\infty\right)$$

I will explain the revised MCP presently, but first some general properties of this tower architecture. A rough idea of the levels of processing is given in figure 12: at each level the processor code is processed by an active process that interacts with it (locally and serially, as usual), but each processor is in turn composed of a structural field fragment in turn processed by a reflective processor on top of it. What I will show is that the implied infinite regress is not problematic, and that the architecture can be efficiently realised, since only a finite amount of information is encoded in all but a finite number of the bottom levels.

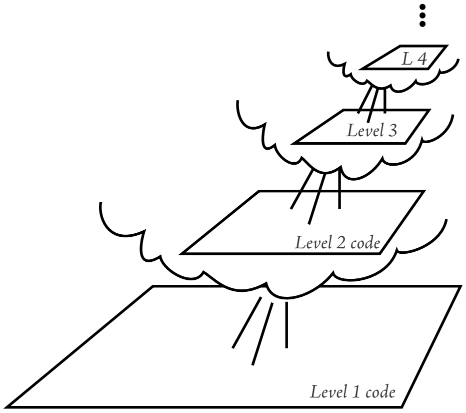


Figure 12 — The 3-Lisp Reflective Tower

There are two ways to think about reflection. On the one hand, on what I will call the "**shifting view**," one can think of there being a primitive and noticeable **reflective act**, which causes the processor, in the sense of the basic locus of animating activity, to shift levels rather markedly either up or down, in what logicians and philosophers might think of as *semantic ascent* and *semantic descent* (this is the explanation that best coheres with some of our pre-theoretic intuitions about reflective thinking, in the sense of contemplation). On the other hand, in what we might instead call the "**tower view**," which accords better with the explanation given in the previous paragraph, the model is instead of an infinite number of levels of reflective processors, each implementing the one below, without any shifting going on.[21] On this tower view, it is not coherent either to ask about *what level the tower is running at*, or to ask how many reflective levels are running: on the tower view *they are all running at once*. The same situation obtains when you use an editor implemented in APL. It is not as if the editor and the APL interpreter are both running together, either side-by-side or independently; rather, the one (the APL interpreter), being "interior" to the other, supplies the anima or agency of the outer one (the editor). To put this another way, when you implement one process in another process, you might want to say that you have two different processes, but you do not thereby have concurrency; the relation is is more like one of part and whole. It is just this sense in which the higher levels in our reflective hierarchy are always running: each of them is in some sense within the processor at the level below, so that it can thereby engender it.

I will not take a principled view on which account—a single locus of agency stepping between levels, or an infinite hierarchy of simultaneous processors—is correct, since they turn out to be behaviourally equivalent. Indeed, one way to characterise the model of reflection being proposed is as the following suggestion:

The semantically cleanest and most elegant way to understand a shifting reflective process is to model it as a tower. **[T]**

(One pragmatic rule of thumb: the simultaneous infinite tower of levels is often the better way to understand *processes*, whereas the shifting-level viewpoint is sometimes the better way to understand *programs*.)

[21] Curiously, there are also intuitions about contemplative thinking, where one is both detached and yet directly present at the same time—which fit more closely with this view.

If we view 3-Lisp on the tower model, as an infinite reflective tower based on 2-Lisp, the code at each level can be understood as like the continuation-passing 2-Lisp MCP presented earlier,[22] but extended in an essential way: to provide a mechanism whereby the user's program can gain access to fully-articulated descriptions of that program's operations and structures. Thus extended, and appropriately located in a reflective tower, I will call this code the **3-Lisp reflective processor procedure (RPP)**. Programs gain reflective access to the articulated descriptions of the program's operations and structures by using what I will call **reflective procedures**—procedures that, when invoked, are: (i) run not at the level at which the invocation occurred, but one level higher, at the level of the reflective processor running the program; and (ii) given as arguments those structures being passed around in the reflective processor. I.e., reflective pro-

[22] "Continuation-Passing 2-Lisp Metacircular Processor" sidebar, page ■■.

Programming in 3-Lisp

For illustration, we will look at a handful of simple 3-Lisp programs. The first merely calls the continuation with the numeral 3; thus a call to it (with no arguments) it is semantically identical to the simple numeral:

```
(define THREE
   (lambda reflect [□ env cant]
      (cant '3)))
```

Thus (THREE) ⇒ 3; (+ 11 (THREE)) ⇒ 14. The next example is an intensional predicate, true if and only if its argument (which must be a variable) is bound in the current context:

```
(define BOUND
   (lambda reflect [[var] env cont]
      (if (bound-in-env var env)
          (cont '$T)
          (cont '$F))))
```

or equivalently

```
(define BOUND
   (lambda reflect [[var] env cant]
      (cant ↑(bound-in-env var env))))
```

Thus (LET [[X 3]] (BOUND X)) ⇒ $T, whereas (BOUND X) ⇒ $F in the global context. The following quits the computation, by discarding the continuation and simply "returning":

cedures are essentially analogues of subroutines to be run "in the implementation," except that:

1. They are written in the same dialect as that being implemented;
2. They can use all the power of the implemented language in carrying out their function—i.e., reflective procedures can themselves make use of further reflective procedures, without limit;[23] and
3. Because they are within, not external to or "underneath" the architecture being implemented, they avoid all of the inelegance, implementation-dependence, and other deleterious

[23] The tower is not a tower of different *languages*. There is a single dialect (3-Lisp) all the way up. What the tower is a tower of is *processors*—necessary because there is different processor state at each reflective level.

```
(define QUIT
    (lambda reflect [[] env cant]
        'QUIT!))
```

There are a variety of ways to implement a THROW/CATCH pair; the following defines the version used in Scheme:

```
(define SCHEME-CATCH
    (lambda reflect [[tag body] catch-env catch-cant]
        (normalise
            body
            (bind tag
                ↑(lambda reflect [[answer] throw-env throw-cont]
                    (normalise answer throw-env catch-cont))
                catch-env)
            catch-cant)))
```

For example:

```
(let [[x 1]]
    (+ 2 (scheme-catch punt
                       (* 3 (/ 4 (if (= x l)
                                     (punt 15)
                                     (- X 1)))))))
```

would designate seventeen and return the numeral 17.

The reflection mechanism is so powerful that many traditional primitives can be defined; LAMBDA, IF, and QUOTE are all non-primitive (user) definitions in 3-Lisp, defined as follows:

aspects of traditional code that has to "reach into the implementation" to do its work.

Reflective procedures are "defined" (in the sense described earlier) using the form

```
(LAMBDA REFLECT ARGS BODY)
```

where ARGS—typically the rail [ARGS ENV CONT]—is a pattern that should match a 3-element designator of, respectively, the argument structure at the point of call, the environment, and the continuation. Some simple examples are given in the "Programming in 3-Lisp" sidebar, above, including a fully functional definition of Scheme's CATCH. Though simple, these definitions would be impossible in a traditional language, since they make crucial access to the full processor state at point of call. Note also that although THROW and CATCH deal explicitly with continuations, the code that uses them need know nothing about such subtleties. More complex routines,

Programming in 3-Lisp (cont'd)

```
(define LAMBDA
   (lambda reflect [[kind pattern bodyJ env contJ
      (cont (ccons kind ↑env pattern body))))

(define IF
   (lambda reflect [[premise then elseJ env contJ
      (normalise premise env
         (lambda simple [premise!]
            (normalise (ef ↓premise! then else) env cant)))))

(define QUOTE
   (lambda reflect [[argJ env contJ (cant ↑arg)))
```

Some comments. First, the definition of LAMBDA just given is, of course, circular; a noncircular but effective version is given in Smith and des Rivières (1984); the one given above, if executed in 3-Lisp, would leave the definition unchanged, except that it is an innocent lie: in real 3-Lisp KIND is a procedure that is called with the arguments and environment, allowing the definition of (LAMBDA MACRO …), etc. CCONS is a closure constructor that uses SIMPLE and REFLECT to tag the closures for recognition by the reflective processor described in section 6. EF is an extensional conditional that normalises all of its arguments; the definition of IF defines the standard intensional version that normalises such as utilities to abort or redefine calls already in process, are almost as simple. In addition, the reflection mechanism is so powerful that many traditional primitives can be defined, rather than having to be provided primitively: LAMBDA, IF, and QUOTE are all non-primitive (i.e., user) definitions in 3-Lisp, again illustrated in the sidebar. A simplistic break package is also presented, to illustrate the use of the reflective machinery for debugging purposes. It is noteworthy that *no* reflective procedures need be primitive; even LAMBDA can be built up from scratch.

The power and simplicity of these examples stems from the fact that the 3-Lisp reflective processor is causally connected in the right way, so as to allow the reflective procedures to run in the system in which they defined, rather than being models of another system. And, since reflective procedures are fully integrated into the system design (their names are not treated as special keywords), they can be passed around in the normal higher-order way. Finally, there is a

only one of the second two, depending on the result of normalising the first. And the definition of QUOTE will yield (QUOTE A) ⇒ 'A.

Finally, we have a trivial break package, with ENV and CONT bound in the break environment for the user to see, and RETURN bound to a procedure that will normalise its argument and pass that out as the result of the call to BREAK:

```
(define BREAK
     (lambda reflect [[argJ env contJ
         (block (print arg primary-stream)
             (read-normalise-print “»”
                (bind* ['env ↑envJ
                       ['cant ↑contJ
                       ['return ↑(lambda reflect [[a2J e2 c2]
                                        (normalise a2 e2 cont))]
                       env)
                primary-stream))))
```

If viewed as models of control constructs in a language being implemented, these definitions will look innocuous; what is important to remember is that they work in the very language in which they are defined.

The 3-Lisp Reflective Processor Program (RPP)

```
1 (define READ-NORMALISE-PRINT
2 .. (lambda simple [level env stream]
3 ..... (normalise (prompt&read level stream) env
4 ........ (lambda simp1e [result]                              ; C-REPLY
5 ........... (block (prompt&reply result level stream)
6 .................. (read-normalise-print level env stream))))))

7 (define NORMALISE
8 .. (lambda simple [struc env cont]
9 ..... (cond [(normal struc) (cont struc)]
10 ......... [(atom struc) (cont (binding struc env))]
11 ......... [(rail struc) (normalise-rail struc env cont)]
12 ......... [(pair struc) (reduce (car struc) (cdr struc) env cont)]))

13 (define REDUCE
14 .. (lambda simple [proc args env cont]
15 ..... (normalise proc env
16 ........ (lambda simple [proc!]                              ; C-PROC!
17 .......... (if (reflective proc!)
18 .............. (□(de-reflect proc!) args env cont)
19 .............. (normalise args env
20 ................. (lambda simple [args!]                     ; C-ARGS!
21 .................... (if (primitive proc!)
22 ........................ (cont □(□proc! . □args!))
23 ........................ (normalise (body proc!)
24 .................................. (bind (pattern proc!) args! (environment proc!))
25 .................................. cont))))))))

26 (define NORMALISE-RAIL
27 .. (lambda simple [rail env cont]
28 .... (if (empty rail)
29 ....... (cont (rcons))
30 ....... (normalise (1st rail) env
31 .......... (lambda simple [first!]                           ; C-FIRST!
32 ............. (normalise-rail (rest rail) env
33 ................. (lambda simple [rest!]                     ; C-REST!
```

sense in which 3-Lisp is simpler than 2-Lisp, as well as being more powerful; there are fewer primitives, and 3-Lisp provides much more compact ways of dealing with a variety of intensional issues (like macros).

8 The 3- Lisp Reflective Processor

3-Lisp is best understood through a close inspection of the 3-Lisp reflective processor—the promised modification of the continuation-passing 2-Lisp metacircular processor mentioned above.

The code for the RPP is presented in a final sidebar, above. NORMALISE (line 7) takes a structure, environment, and continuation, and: (i) returns the structure unchanged (i.e., sends it to the continuation) if it is in normal form; (ii) looks up the binding if it is an atom; (iii) normalises the structure's elements if it is a rail;[24] and (iv) otherwise reduces the `CAR` (procedure) with the `CDR` (arguments). `REDUCE` (line 13) first normalises the procedure, with a continuation (`C-PROC!`) that checks (line 17) to see whether it is reflective.[25] If it is not reflective, `C-PROC!` normalises the arguments, with a continuation that either expands the closure (lines 23–25) if the procedure is non-primitive, or else executes it directly (line 22) it if it is primitive.

As an example, consider `(REDUCE '+ '[X 3] ENV ID)`, assuming that `X` is bound to the numeral `2` and `+` to the primitive addition closure in `ENV`. At line 22, `PROC!` will designate the primitive addition closure, and `ARGS!` will designate the normal-form rail `[2 3]`. Since addition is primitive, we must simply do the addition. `(PROC! . ARGS!)` would not work, because `PROC!` and `ARGS!` are at the wrong level; they designate *structures*, not functions or arguments. For a brief instant, therefore, we dereference them (with ↓), do the addition, and then regain our meta-structural viewpoint with ↑.[26] If the procedure is

[24] `NORMALISE-RAIL` is 3-Lisp's tail-recursive continuation-passing analogue of Lisp 1.5's `EVLIS`.

[25] I adopt a convention of using exclamation point suffixes on atom names used as variables to designate normal form structures.

[26] One way to understand this is to realize that the reflective processor simply asks its processor to do any primitives that it encounters—i.e., it passes responsibility for the execution of primitives up to the processor running it. In other words, each time one level uses a primitive, its processor runs around setting everything up, finally reaching the point at which it must simply *do* the primitive action, whereupon it asks its own processor for help. But, of course, that processor—i.e., the processor running the processor in question—will also come racing towards the edge of the same cliff, and will similarly duck responsibility, handing the primitive up yet another level.

The net result, from the "tower" perspective, is that every primitive ever executed is handed all the way to the (infinitely remote) top of the tower. There is then a magic moment, when the thing actually happens—and then the answer filters all the way back down to the level that started the whole procedure. It is as if the *deus ex machina*, living at the top of the tower, sends a lightning bolt down to some level or other, once every intervening level gets appropriately lined up (rather like the sun, at Stonehenge and the Pyramids, reaching down through a long tunnel at just one particular moment during the year).

reflective, however (line 18), it is called directly, not processed, and given the obvious three arguments (ARGS, ENV, and CONT) that are being passed around. ↓(DE-REFLECT PROC!) is merely a mechanism to "purify" the reflective procedure so that it does not reflect again, and to dereference it to be at the right level (we want to *use*, not *mention*, the procedure designated by PROC!). Note that line 18 is the only place that reflective procedures can ever be called; this is why they must always be prepared to accept exactly those three arguments.

This leads to an important point:

Reflective processor program
line line 18 is the essence of 3-Lisp.

Line 18 alone engenders the full reflective tower, for it says that some parts of the object language—the code processed by this program—are called directly *in* this program. It is as if an object level fragment were included directly in the meta language, which raises the question of who is processing the meta language. This is where the tower enters the picture: the claim underlying 3-Lisp is that an *exactly equivalent reflective processor* is processing this code, too—and that this fact can be true without vicious threat of infinite ascent.

The result is to allow a reflective procedure "to be executed in the middle of the processor context." It is handed, as arguments, environment and continuation structures that designate the processing of the code below it, but it is *run* in a different context, with its own (implicit) environment and continuation, which are in turn represented in structures passed around by the processor one level above it. In this way a reflective procedure is given causal access to the state of the process that was in progress (answering one of the three initial requirements for reflection); as a result, it can cause any effect it wants since it has complete access to all future processing of that code. Furthermore, it has a safe place to stand, where it will not

Except, of course, that nothing ever happens, ultimately, except primitives. In other words the enabling agency, which must flow down from the top of the tower, consists of an infinitely dense series of these lightning bolts, with something like 10% of the ones that reach each level being allowed through that to the level below (and then 10% of those reaching to the level below it, etc.).

All infinitely fast.

«This should be edited to refer to the Implementation paper.»

conflict with the code being nm below it (thereby meeting the third criterion).

These various protocols illustrate a general point. As mentioned at the outset, part of designing an adequate reflective architecture involves a trade-off between being so connected that one steps all over oneself (as in traditional implementations of debugging utilities), and so disconnected (as with metacircular processors) that one has no effective access to what is going on. The suggestion made here is that the 3-Lisp reflective tower provides just the right balance between these two extremes, solving the problem of vantage point as well as of (both directions of) causal connection.

The 3-Lisp reflective processor unifies three traditionally independent capabilities in Lisp: (i) the explicit availability of EVAL and APPLY, (ii) the ability to support metacircular processors, and (iii) explicit operations provided for debugging purposes (such as MacLisp's RETFUN and Interlisp's FRETURN[27]). It is striking that the latter facilities are required in traditional dialects, in spite of the presence of the former, especially since they depend crucially on implementation details, violating portability and other natural aesthetics. In 3-Lisp, in contrast, all information about the state of the processor is fully available within the language itself—suggesting that its reflective architecture constitutes something of an appropriate theoretical unification of the kinds of extension that have heretofore had to be made in ad-hoc and non-transportable ways.

9 Threats of Infinity and Finite Implementations

The argument as to why 3-Lisp is finite is complex in detail, but simple in outline and substance. In brief: the proof relies on showing that the reflective processor is tail-recursive in two senses:

1. It runs programs tail-recursively, in that it does not build up records of state for programs across procedure calls (only on argument passing); and
2. It itself is fully tail-recursive, in the sense that all recursive calls within it (except for unimportant subroutines) occur in tail-recursive position.

[27] «Refs?»

As a result, the reflective processor can be executed by a simple finite state machine. In particular—and this is the crucial point—it can run itself without using any state at all. Once the limiting behaviour of an infinite tower of copies of this processor has been determined, therefore,[28] that entire chain of processors can be simulated by another finite state machine, of complexity only moderately greater than that of the reflective processor itself.[29] A full copy of such an implementing processor[30] and a much more substantive discussion of tractability is provided in Smith & des Rivières (1984).

10 Conclusions and Morals

The use of Lisp as a language in which to explore programming semantics and reflection is not essential; the ideas should hold in any similar circumstance. I have chosen Lisp because it is familiar, because it has rudimentary self-referential capabilities, and because there is a standard procedural self-theory (continuation-passing metacircular "interpreters"). Work has begun, however, on designing reflective dialects of a side effect-free Lisp and of Prolog, and on studying a reflective version of the ⋹calculus (the last being an obvious candidate to be used as a basis for a mathematical study of reflection).[k]

The techniques used here to define 3-Lisp can be generalised rather directly to these other languages. As suggested at the outset, in order to construct a reflective dialect one needs:

1. To formulate a theory of the language analogous to the metacircular processor descriptions we have examined;
2. To embed this theory within the language; and
3. To connect the theory with the underlying language in an appropriate causally connected way—i.e., so as to allow for

[28] This has not yet been explained in this paper; see «refer to the implementation paper.»

[29] It is an interesting open research question whether that "implementing" processor can be algorithmically derived from the reflective processor code.

«Note that this has yet to be done … »

[30] Consisting (including all utilities) of only about 200 lines of 2-Lisp code.

[k] «May put in a sidebar on the result? I have it somewhere…»

> both "upwards" and "downwards" connection—by allowing reflective procedures invocable in the object language the ability to run (non-reflectively) in the processor (as was done in line 18 of the 3-Lisp reflective processor program).

It remains to implement the resulting infinite tower; a discussion of general techniques, which again would readily generalize to languages other than 3-Lisp, is presented in des Rivières and Smith (1984).

It is partly a consequence of using Lisp that I have used non-data-abstracted representations of functions and environments; this facilitates side effects to processor structures without introducing unfamiliar machinery. It is clear that environments could be readily abstracted, although it would remain open to decide what modifyonlylling operations would be supported (changing bindings is one, but one might wish to excise bindings completely, splice in new ones in, etc.). In standard l-calculus-based metatheory there are no side effects (and no notion of processing); environment designators must therefore be passed around ("threaded") in order to model environment side effects. It should be simple to define a side effect-free version of 3-Lisp with an environment-threading reflective processor, and then to define SETQ and other such routines as reflective procedures. Similarly, I have assumed in 3-Lisp a single structural field commonly visible from all code; one could define an alternative dialect in which the structural field, too, was threaded through the processor as an explicit argument, as in standard metatheory.

The representation of procedures as closures is troublesome.[31] I would be the first to admit that 3-Lisp provides too fine-grained (i.e., too metastructural) access to function designators—including continuations and the like. Given an appropriately abstract notion of procedure, it would be natural to define a reflective dialect that used abstract structures to encode procedures, and then to define reflective access in such terms. While I did not follow this direction here, in order to avoid taking on another very difficult problem, another intent of future work is to move in this direction.

[31] Closures are failures, in a sense, in that they encode far more information than should be required in order to identify a function in intension; the problem being that we do not yet know what a function in intension might be.

These considerations all illustrate a general point: in designing a reflective processor, one can choose to bring into view more or less of the state of the underlying process. Fundamentally, it reduces to a design choice of what one wants to reify or make explicit, and what one wants to absorb. As currently defined, 3-Lisp reifies (i) the environment and (ii) the continuation, thereby making explicit those two implicit dimensions of processing one level below. It absorbs (iii) the structural field and (iv) the global environment; in addition, as mentioned earlier, it completely absorbs (v) the animating agency of the whole computation. If one were to define a reflective processor based on a metacircular processor that also absorbed the representation of control (in the style of the non-continuation-passing 2-Lisp MCP,[32] which embedded the control structure of the code being processed with the control structure of the processor), then reflective procedures would not have access to, and therefore could not affect, a base program's control structure. In any real application, it would need to be determined just what parts of the underlying dialect required reification.

More interestingly, one might be able to design a reflective language in which individual reflective procedures could specify, with respect to a very general meta-theory, which aspects they wanted explicit access to (simply environment in one case, animating agency in another, control structure but not agency in a third, etc.). In such a design, operations that needed only environmental access, such as BOUND?, would not need to traffic in continuations. While a modification of 3-Lisp that provides such "contextually optional" access to environment, continuation, and structural field, a full exploration of this possibility remains for future work.

One final point. Throughout this paper I have talked about semantics, but I have so far presented no mathematical semantical accounts of any of the dialect presented. To do so for 2-Lisp is relatively straightforward (see des Rivières and Smith (1984)[l]), but it remains to develop appropriate semantical equations to describe 3-Lisp. While might initially be tempting to construct such a model

[32] Sidebar on p. ■■.

l) «Check; not sure this was ever done? Was it in the manual?»

based on the implementation strategy described in des Rivières and Smith (1984), I believe that doing so would be a failure. Instead, what is needed is a two-step process:

1. To construct a mathematical account of the "infinite tower" view of 3-Lisp—i.e., to take the limit as $n \rightarrow \infty$ of 2-Lisp$_n$, as suggested in §■■; and then
2. To prove, in terms of that model, that the finite implementation strategies presented in des Rivières and Smith (1984) are *correct*.

This awaits further work. Additional future work would include: (i) exploring what it would be to deal explicitly, in the semantical account, with anima or agency (rather than simply absorbing it), which would introduce parallelism into the reflective act; and (ii) formulating a more general account of the requisite causal connection, that are so crucial to the success of any reflective architecture. These various tasks will require more radical reformulations of semantics than have been considered here.

Acknowledgements

I have benefited greatly from the collaboration of Jim des Rivières on these questions, particularly with regard to issues of effective implementation. The research was conducted in the Cognitive and Instructional Sciences Group at the Xerox Palo Alto Research Center (PARC), as part of the Situated Language Program of Stanford's Center for the Study of Language and Information (CSLI).

References

Batali, John, "Computational Introspection," Artificial Intelligence Laboratory Memo AIM-TR-701, Massachusetts Institute of Technology, Cambridge, MA, 1983.

des Rivières, Jim and Smith, Brian Cantwell, "The Implementation of Procedurally Reflective Languages," 1984 Conference on LISP and Functional Programming, Austin, Texas, August 1984. Also available as Xerox Palo Alto Research Center (PARC) Report ISL–4, Palo Alto, CA (1984) and Stanford Center for the Study of Language and Information Report CSLI-84-9 (1984). Reprinted here as Chapter ■■.

Doyle, Jon, "A Model for Deliberation, Action, and Introspection," Artificial Intelligence Laboratory Memo AIM-TR-581, Massachusetts Institute of Technology, Cambridge, MA, 1980.

Fodor, Jerry. "Methodological Solipsism Considered as a Research Strategy in Cognitive Psychology," *The Behavioural and Brain Sciences*, 3:1 (1980) pp. 63–73; reprinted in Fodor, Jerry, *Representations*, Cambridge, MA: Bradford, 1981.

Genesereth, Michael and Lenat, Douglas B., "Self-Description and Modification in a Knowledge Representation Language," Heuristic Programming Project Report HPP-80-10, Stanford University Department of Computer Science, 1980.

McCarthy, John et al., *lisp 1.5 Programmer's Manual*. Cambridge, MA: MIT Press, 1965.

Smith, Brian Cantwell, *Reflection and Semantics in a Procedural Language*, Laboratory for Computer Science Report MIT-TR-272, 1982. Abstracts, Prologue, and Chapter 1 reprinted here as Chapter ■■.

Smith, Brian Cantwell and des Rivières, Jim, "Interim 3-Lisp Reference Manual," Report ISL-1, Xerox Palo Alto Research Center (PARC), Palo Alto, CA (1984…■■).

Steele, Guy, "LAMBDA: The Ultimate Declarative," Artificial Intelligence Laboratory Memo AIM-379, Massachusetts Institute of Technology, Cambridge, ma, 1976.

Steele, Guy and Sussman, Gerald, "The Revised Report on SCHEME, a Dialect of LISP," Artificial Intelligence Laboratory Memo AIM-452, Massachusetts Institute of Technology, Cambridge, ma, 1978a.

Steele, Guy and Sussman, Gerald, "The Art of the Interpreter, or, The Modularity Complex (parts Zero, One, and Two)," Artificial Intelligence Laboratory Memo AIM-453, Massachusetts Institute of Technology, Cambridge, ma, 1978b.

Weyhrauch, Richard W., "Prolegomena to a Theory of Mechanized Formal Reasoning," *Artificial Intelligence* 13:1,2 (1980) pp. 133–170.

2010 Perspective (cont'd)

of the MIT Artificial Intelligence Laboratory, where I was enrolled.

Given the impossibility of bringing Mantiq to fruition, it was fortunate that 3-Lisp and procedural reflection were able to serve as the focus of a completable doctoral dissertation—though the advertising was disingenuous, since although Mantiq was genuinely supposed to be reflective, 3-Lisp ultimately amounted to being only what I would later call "introspective."[6] (Mantiq was also intended to be descriptively as well as procedurally reflective; though I did recognize that 3-Lisp was limited to the procedural case.)

Some of the history of Mantiq and 3-Lisp is described in the Preface to the dissertation that resulted, published as a technical report under the name "Reflection and Semantics in Procedural Languages" (RSPL), *q.v.*[7] Of special relevance here is the fact that the semantic orientation adopted in the 3-Lisp design, according to which programs are taken as *effective ingredients within computational processes*, rather than as external specifications of (or prescriptions for) them, was more familiar within knowledge representation (KR) and AI circles than it was in the programming language community per se. This perspective, which I dub an "ingrediential" view of programs, derives in part from the fact that I came to the Mantiq project out of an interest in knowledge representation, and that the KR community conceives its task as one of developing computer analogues of the mental structures that underlie active, real-world knowledge and thought processes—i.e., *as they occur during the course of a person's (or system's) ongoing life*—rather than as statically or once-and-for-all "specifying a mind," in the way that one might take to be the task of DNA. This ingrediential stance to reflection is quite explicit in RSPL, for example in the discussion of what I called the "Reflection Hypothesis":[8]

> *In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process can be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures.*

At the time this 3-Lisp paper was published, I did not appreciate the theoretical significance, especially as regards semantics, of viewing programs from different perspectives. Recognition began to dawn soon thereafter, when I encountered the incomprehensibility with which my programming language colleagues greeted my approach to 2-Lisp (and thus 3-Lisp) semantics. A particularly telling event occurred in 1984, when—proud of what I took to be its semantical cleanliness—I invited Joseph Goguen and Jose Meseguer, programming language theorists at SRI, to sketch out a "formal denotational semantics" for 2-Lisp. My plan was to use what they developed as a basis for initiating a mathematical analysis of 3-Lisp and reflection. When they generously came back with a proposal, however, I was—to be frank—astonished. What they took to be a mathematically clean semantical analysis obliterated what I took to be essential to 2-Lisp's semantical clarity—conflating distinctions I had taken such pains to maintain, such as among handles, numerals, and numbers, and between sequences and rails. Entities I took to be concrete were treated as abstract; the grounds on which I had rested my critique of the Lisp conception of evaluation had vanished; and in general their "theoretically clean" version of 2-Lisp had undergone a transformation that not only rendered it wholly unfamiliar to me, but that "disappeared" what was—at least in my eyes—its major contribution. Needless to say, , the proposed collaboration stalled, in spite of great respect on both sides (I mean nothing indicting by telling this tale; we were simply approach what we took to be a common subject matter from radically different perspectives). I never did develop a mathematical account of reflection—nor, to my knowledge, has anyone else.

Fortunately, in spite of this setback, the work on 3-Lisp and procedural reflection itself was kindly received in the larger community. After this paper appeared at the Principles of Programming Languages conference (POPL) in 1984, interest in reflection burgeoned around the world, and a variety of reflection conferences were held over the subsequent 10 years.[9]

But the issues that had surfaced in the interaction with Goguen and Meseguer were a harbinger of more profound intellectual challenges than at the time I knew how to resolve. I had staked my dissertation on the fundamental thesis on which 3-Lisp is based (thesis [R], §1, p. (■■): *that reflection is relatively straightforward, if implemented on a semantically sound base.* While, in an overall sense, the topic of procedural reflection was widely picked up, that orienting thesis, with no exceptions of which I am aware, was resoundingly ignored.[10] At first I was puzzled by people's blindness to or even dismissal of it,[11] but I gradually came to appreciate that the incomprehensibility of this semantical thesis rested on the considerable conceptual difference of viewing programs as ingredients in computational processes, rather than as specifications or prescriptions of them.

As one would expect, the clearer I became on the underlying issues, the more I was able—especially in conversation—to explain the perspective from which 3-Lisp was designed. As I quickly learned, however, success in describing its architectures by and large required that I *not* use the ingrediential vocabulary I am employing here—i.e., depended on my not saying that the two dialects were based on a view of programs as causally effective process-internal ingredients. Rather, I had to describe them from a viewpoint that at the time felt alien to me: taking programs to be external, if nevertheless effective, process specifications or descriptions (or even prescriptions). A conversation with Gordon Plotkin (again in the mid 1980s) at Stanford's Center for the Study of Language and Information (CSLI) is illustrative. After failing to communicate anything about what mattered to me about 2-Lisp using my own terminology, I attempted to adopt his—i.e., tried to "inhabit" the specificational view—and said that what I was interested in was "*the semantics of the semantics of programs*." The ploy must have worked, as I recall him nodding and smiling. But the differences remained profound, and nothing further came of the conversation. Although I made some subsequent attempts to explain the differences in viewpoints (e.g., in (■■), it seems safe to say that the 2-Lisp and 3-Lisp approach to semantical clarity—and the idea of theorizing distinct procedural and declarative aspects of program meaning—was met with virtual silence when first presented, and then quickly faded into the background.

Over the intervening 25 years I have developed a much deeper understanding of these communicative failures, as well as an appreciation of the intellectual history that gave rise to them. The issues lie deep in the foundations of computing, and derive in part from the ways in which computer science has taken over technical terminology from philosophical and mathematical logic, but has used it for different purposes. Of numerous issues, one looms large in the present context: for reasons traceable as far back as Turing's original 1937 paper, computer scientists in general, and programming language theorists in particular, use what a classical logician would consider semantical vocabulary and model-theoretic techniques to analyse what that same logician would think of as fundamentally syntactic and/or proof-theoretic concerns. Disentangling this history helps to clarify all manner of communicative failures, theoretical confusions, and contextually incomprehensible behaviours—including such seemingly diverse topics as misunderstandings (on all sides) of Searle's Chinese Room thought experiment, the widespread use of constructive mathematics and intuitionistic logic in theoretical computer science (such as Martin-Löf's intuitionistic type theory) the structure of reflection, the meteoric rise in popularity (perhaps even the provenance) of Girard's linear logic,[12] and the substantial distraction we have all suffered, in my view, from focusing exclusively on the semantics of *programming languages*, rather than on the semantics of *individual programs*.

Elsewhere I have made some stabs at explaining these issues,[13] but only briefly, and in passing. One of the goals of *The Age of Significance* (AOS) project,[14] being launched as this is being written, is to spell out this history in ways that facilitate understanding across the boundaries of computer science—both "externally," as it were, by allowing what matters about computing to be understood from an external intellectual perspective, and "internally," by enabling the genuine semantical insights of the logical tradition to be appreciated within computer science (something that in my opinion has largely not yet occurred).

My exploration of these foundational issues has primarily taken place in my investigations into the philosophy of computing, and will be reported on as such. More technically, after the publication of this paper my attention did not stay focused on programming languages, but turned back towards the issues that had originally motivated Mantiq: how to generalize the lessons learned here in the context of people and/or systems able to reason about the concrete, external world.

I was sobered not only by the daunting challenges of doing justice to real-world metaphysics and ontology, but also by an inadvertent lesson gained from the 3-Lisp exercise: the untenable pedantry of excessive semantical strictness. Not only was it manifest that dealing with real-world ontology was a profoundly more serious challenge than anything for which the 3-Lisp project provided preparation, but it also quickly became clear that semantics itself, *and any ideal of "semantical clarity,"* would have to be rethought in the most fundamental way, if we were even to approach, in artificial systems, the prowess and facility with which we people think about and find intelligible the worlds in which we are embedded. Some initial steps in these directions were reported in "The Correspondence Continuum" and "Varieties of Self-Reference," both written in 1986.[15] But as noted in the annotations to those papers included in this volume, I ultimately came up against what I came to call an "ontological wall,"[16] prompting me to delve even deeper into epistemology and metaphysics—a shift in emphasis that led to the writing of *On the Origin of Objects* (O3) in 1996,[17] and that continues to this day.

I do not believe it would be impossible to incorporate at least some of the lessons of O3 in a reflective computational system—in part because of not believing that 'computational' is a restrictive property (see AOS). But until such a day—a day that it is hard to know whether I myself will ever reach—the original motivations for developing 3-Lisp, the fundamental insights on which it is based, and the original vision of Mantiq all remain waiting in the wings.

Notes

[1] Sidebars and footnotes with text in sans-serif font, as in this case, contain comments and reflections added in 2010, rather than material that appeared in the original paper.]

[2] 'Mantiq' (منطق) is roughly the Arabic equivalent of the Greek *logos* (ǂǂǂǂǂ)—meaning *speech*, *manner of speaking*, *eloquence*, or *logic* «ref: *The Hans Wehr Dictionary of Modern Written Arabic*). It is best known in the title *Mantiq al-Tayr* (منطق الطير), a book of poems by the Sufi poet Farid al-Din Attar, sometimes translated as "The Language of the Birds" but more commonly as "The *Conference* of the Birds."

[3] At least what philosophers would call its "narrow" meaning (cf. «ref»). Not only did I quickly come to realise that a great variety of different things been called the "meaning" of an expression or idea, over the years, but I have also come to believe there never will be a "final catalogue" of just which of the infinite number of aspects of an intentional utterance or event can or do matter to its full significance. Even more challenging, from a design point of view, I believe that what we take to be the "meaning" of such any such event or occassion (let alone what "type" it instantiates) is likely contextually dependent not only on facts about the event so taken, but on the circumstances of the situation in which the meaning is referred to.

Moreover, whatever eventual story about meaning one were to adopt, it is likely that a true "fusion" of meaning and structural identity would prove impossible in the limit, since it is usually possible, given any such view, to construct examples showing that meaning identity is uncomputable. Still, having some such goal as an ideal can provide motivation and direction towards "higher-level" archictures of intentional capacity.

[4] The first drafts of the report on 3-Lisp were designed to be chapter 13 of the infeasible Mantiq dissertation.

[5] The idea can clearly be generalised, allowing one to "step sideways," as it were, so as to be able to see one whole tower as a unity, etc. But I say "first good idea" because I was interested in a much more radical kind of reflection, involving a wholesale "leap" across a chasm from one locus of intelligibility to another, which (by definition) cannot be "viewed" from a vantage point accessible within the "prior" epistemic architecture. The merest sketch of such an idea is mentioned in O3 «ref»;

I plan to explore it much more fully in Phase II of AOS «ref».

[6] See “Varieties of Self-Reference,” Chapter ■■.

[7] Reprinted here as chapter ■■. The dissertation itself was submitted as "Procedural Reflection in Programming Languages'; the change in title reflected not only the importance of thesis [R] (p. ((), but also my increasing awareness of the importance of the semantical model on which the reflective architecture was based.

[8] Op. cit, pp. ■■.

[9] «References»

[10] For example, although the Wikipedia web page on reflection in computer science (below) credits the 3-Lisp work as introducing the notion of reflection into programming languages, it makes no mention of the rationalised semantics on which the 3-Lisp design was based (in spite of discussion throughout the article about the “subject matter” of programming constructs). Similarly, none of the ten examples of reflection in contemporary languages presented at the end of the article are designed in terms of an explicit theorization of subject matter or declarative import.

http://en.wikipedia.org/wiki/Reflection_(computer_science)

[11] Cf. Daniel P. Friedman and Mitchell Wand, “Reification: Reflection without Metaphysics,” LISP and Functional Programming Conference, 1984, pp 348-55.

[12] «References»

[13] E.g., in “The Foundations of Computing,” reprinted here as chapter ■■.

[14] See http://www.ageofsignificance.org

[15] See chapter ■■ and chapter ■■.

[16] E.g., see “The Foundations of Computing,” reprinted here as chapter ■■.

[17] On the Origin of Objects, MIT Press, Cambridge, MA: 1996.

4 — The Implementation of Procedurally Reflective Languages[†]

Jim des Rivières and Brian Cantwell Smith[*]
IBM and University of Toronto

Abstract

In a procedurally reflective programming language, all programs are executed not through the agency of a primitive and inaccessible interpreter, but rather by the explicit running of a program that represents that interpreter. In the corresponding virtual machine, therefore, there are an infinite number of levels at which programs are processed, all simultaneously active. It is therefore a substantial question to show whether, and why, a reflective language is computationally tractable. We answer this question by showing how to produce an efficient implementation of a procedurally reflective language, based on the notion of a level-shifting processor. A series of general techniques, which should be applicable to reflective variants of any standard applicative or imperative programming languages, are illustrated in a complete implementation for a particular reflective LISP dialect called 3-LISP.

[†] Slightly revised version of a paper that appeared in the *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming.* Also published as Intelligent Systems Laboratory Report No. ISL-4, Xerox Palo Alto Research Center, July 1984; and as Report No. CSLI-84-9, Stanford University Center for the Study of Language and Information, July 1984.

[*] Coach House Institute, Faculty of Information, University of Toronto 90 Wellesley St W, Toronto, Ontario M5S 1C5 Canada

2010 Perspective[α1]

——— to be written ———

Notes

[α1] Sidebars and footnotes with text in sans-serif font, as in this case, contain comments and reflections added in 2010, rather than material that appeared in the original paper.

1 Introduction

As described in (Smith 82a; Smith 84), a reflective computational system is one in which otherwise implicit aspects of the system's structure and behaviour are available for explicit inspection and manipulation. A procedurally reflective programming language is a particular architecture for reflection in which all programs are executed not through the agency of a primitive and inaccessible interpreter, but rather by the explicit running of a program that represents that interpreter. Since the latter program, which we call the **reflective processor program** (**RPP**),[1] is written in the same reflective language as the user program, it too must be executed by the explicit running of a copy of itself. And so on ad infinitum. In the abstract or virtual machine, in other words, *no* program is ever run directly, but instead is run indirectly through the explicit action of the running of the RPP.

In the virtual machine, therefore, there are an infinite number of reflective **levels** at which programs are processed, all simultaneously active (in exactly the same way that a traditional program

[1] We use 'processor' in place of 'interpreter' in order to avoid confusion with the semantic (model-theoretic) notion of interpretation. See (Smith 1982a and (Smith 1984).

written in some language L and the program that implements language L are simultaneously active). Each level has its own local state distinct from the state of neighbouring levels (i.e., there is one "control stack" per level). The architecture resembles an infinite tower of continuation-passing metacircular interpreters,[2] except that (again as discussed in (Smith 84) there are crucial *causal connections* between the levels. Specifically, a program running at one level can provide code to be run at the next higher level—i.e., at the level of the original program's processor—thereby gaining *explicit* access to the formerly *implicit* state of the computation.

The situation is analogous to one where a user program is allowed to insert code into the implementation, except that in the reflective case the implementation is written in the same language as the original user program. This facility enables the user to define new control constructs, implement debuggers, etc., without requiring special hooks into the *actual* implementation. The technique is so powerful that large classes of control structures can be straightforwardly defined in a reflective language in terms of primitive data-manipulation procedures.

We believed that reflection is an important tool to add to any language designer's toolbox. Even if one decides that reflection is too powerful to make generally available to users, a designer may find that the task of producing a correct and complete implementation (e.g., including debugging facilities) is simplified by adopting a reflective architecture as an underlying model. As this paper will show, the issues that arise in implementing a simple reflective language are remarkably similar to the issues that arise in implementing complex non-reflective languages containing primitive debugging facilities and fancy control constructs. Also, reflection has interesting (and unique) properties that are a direct effect of making it possible to view a computation from more than one vantage point at the same time. For example, a purely functional procedurally reflective language, entirely lacking side effects in its primitive functions or special constructs, can nevertheless use reflection to define an assignment statement.[3] In general, reflection

[2] McCarthy (1965), Steele & Sussman (1978b).

[3] Exactly the same principle is employed when giving a denotational semantic account of a programming language that has assignment statements: the

is a technique whereby a theory *of* a language embedded *within* a language can convey otherwise unavailable power.

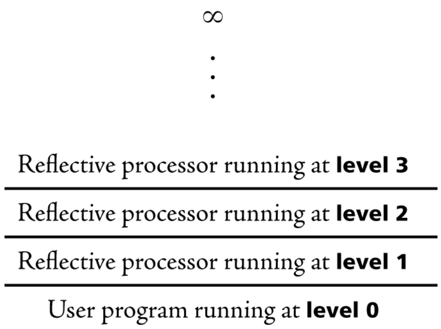

∞
.
.
.
Reflective processor running at **level 3**
Reflective processor running at **level 2**
Reflective processor running at **level 1**
User program running at **level 0**

Figure 1 — Levels in a reflective tower

Given a virtual machine consisting of an infinite number of levels of processing, it is clear that one of the most important questions to ask about a reflective language is whether, and why, it is computationally tractable. This paper addresses that problem by considering the general question of producing an efficient actual implementation of a procedurally reflective language. We show, in other words, how to construct a finite program to simulate an infinite tower of reflective levels. After presenting general principles and techniques that should apply to reflective variants of any standard applicative or imperative programming languages, we present an efficient implementation of a particular reflective LISP dialect called 3-LISP.[4]

2 Towers of Processing

We start by numbering each reflective level: 0 for the level at which the user's program is processed, 1 for the level at which the program that runs the user's program is processed, and so on. In general, the structures (programs and data and so forth) at any given level represent the state of the computation one level below; thus level $n+1$ is one level "meta" to level n.[5] This arrangement, which we call a **tower,** is depicted in figure 1. Finite heterogeneous towers of processing (i.e., a finite number of different languages) are commonplace—a LISP program running at level 0, run by the LISP processor (interpreter) which is a machine language program running at level 1, which, in turn might be run by an emulator, a

state of the computation that was implicit at the level of the program is made explicit at the level of the mathematical metalanguage in which the account of the language is formulated.

[4] Smith (1984), Smith & des Rivieres (1984).

[5] Though it is not quite required by the underlying notion, it is natural to have structures at one level designate (name) structures at the level below. Again, see (Smith 1982a) and (Smith 1984).

microcode program running at level 2.[6] What distinguishes procedurally reflective architectures is that the processing tower is infinite and homogeneous. The user's program (at level 0) is run by the RPP (running at level 1), which is in turn run by another incarnation of that same RPP (at level 2). And so on.[7]

The claim that a user's program runs at level 0 is in fact a lie: the whole point of procedurally reflective languages is to allow user code also to run at level 1 or higher, thereby giving user programs explicit access to the data structures encoding their own state, and therefore power to direct the course of their own computation. What we are calling the **actual** implementation (that process that mimics the virtual infinite tower) must therefore be able to provide *explicit structures encoding the otherwise implicit state of the user's program at any arbitrary level.* It is this crucial fact that makes procedurally reflective systems more difficult to implement than systems without such "introspective" capabilities.

The first step in providing such an implementation is to discharge the threat of the infinite. The key observation is that the activity at most levels—in fact at all but a finite number of the lowest levels—will be monotonous: the RPP will primarily be used to process the same old expressions, namely those that make up the RPP itself. From some finite level *k* all the way to the "top," in other words, the tower will just consist of the processor processing the processor. Identify as **kernel** those expressions in the RPP that are used in the course of processing the RPP which is running one level below.[8] Call a processing level **boring** if the only expressions that are proc-

[6] In a finite tower, there is one level which is run "by the hardware", at which point there is no further program, and therefore no question of who runs it. See (Smith 1982b).

[7] Throughout, we assume that a level implements the level below it, so the sense of direction is opposite from common practice, where one normally thinks of an implementation of a language as being *below* the language implemented. Our usage, however, is in line with the customary view that a name or designator is *above* the referent or designation (see note ■■).

[8] There are three classes of expressions that one might think of as the relevant base for the induction: those that are *primitive*, those that are *simple* (i.e., do not involve reflection), and those that are *kernel*. In 3-LISP the three classes overlap but are distinct; as discussed in §4d, it is the kernel ones that are key to a correct implementation.

essed at that level (in the course of a computation) are kernel expressions. Define the **degree of introspection** (Δ) of a program to be the least *m* such that when the program is run at level 0, all levels numbered higher than *m* are boring.

All programs consisting entirely of kernel expressions have Δ=0. Normal programs (i.e., standard user programs that do not use any reflective capabilities) will have Δ=1, meaning that no out-of-the-ordinary processing is required at level 1. The processing of the level 0 program, in other words, will not entail running non-kernel code at level 1. Δ=2 would be assigned to programs that involve running non-kernel user code at levels 0 and 1, but not at the second reflective level. And so on. Just as a correct implementation of recursion is not required to terminate when a procedure recurses indefinitely, a correct implementation of a procedurally reflective system need terminate only on computations having a *finite* degree of introspection. Tractable reflective programs, in other words, are those with a finite degree of introspection (Δ).

We can now formulate a general plan for implementing a procedurally reflective system. Suppose that one has an implementation processor G (a real, active, processor—not just a program for a processor) that engenders the behaviour of the processor for the language *provided that the program it is given to run has* Δ=1. The existence of such a G is a reasonable presumption, since G is essentially just a processor for the language in question stripped of its reflective capabilities. A procedurally reflective language minus the ability for the user to use reflection is likely to be conventional. 3-LISP minus reflection, for example, is a simple SCHEME-like language that will succumb to standard implementation techniques.[9]

Given G, we can show why any reflective program is tractable by induction. The crucial observation is that the overall degree of introspection (Δ) of an RPP that is running some Δ=*n* program is itself Δ=*n*−1 (this follows directly from the definition of Δ). So, if instead of having the user program run directly by G, it is run indirectly by the RPP which itself is run directly by G, then any Δ=2 user program will be processed correctly. In general, any Δ=*n* program can be run correctly by G provided that *n*−1 levels of genuine

[9] See for example Allen (1978), Steele (1977a), and Henderson (1980).

RPP are placed in between. This result is depicted in figure 2.[10]

Since it is unlikely that a program's Δ can be determined without processing it, the tractability argument just given does not lead directly to a very useful implementation strategy. But based on its insight, we can design a series of implementations, the final version of which is actually reasonably efficient.

Implementation processor G running at **level n**

Δ=**1** user program/RPP running at **level n–1**

⋮

Δ=**n–3** user program/RPP running at **level 3**

Δ=**n–2** user program/RPP running at **level 2**

Δ=**n–1** user program/RPP running at **level 1**

Δ=**n** user program running at **level 0**

Figure 2 —Running a Δ=n program with a processor that can only handle Δ=1

The first approach is simply to start out with G running at some level, and then to restart the computation at the beginning with G at a higher level if the previous try does not succeed (specifically: if it fails because of encountering a reflective request). More formally, assume initially that Δ=1, and give the program to G to run directly. If G detects that the program that it is running has Δ>1, start the whole computation over again, but this time run the user program indirectly, with one more level of intervening RPP. Repeat this last step until G does not protest. This procedure is guaranteed to terminate for any computation with a finite degree of introspection; it requires only that G be able to recognize, *at some point during its processing*, that a computation has a Δ>1, and that the computation be re-startable.[11] Both of these assumptions are theo-

[10] We talked previously only about a *program's* running at a given level; after introducing G we have described it—an active process, not a program—as running at some level as well. The relationship is this: if we say that G is running at level *k*, we mean that a program at level *k* is being run by G directly, without the intervention of any higher levels of RPP.

[11] The re-startability of a computation does not imply that external world side effects (e.g., input/output) would be out of the question for a procedurally reflective system run in this way. All that would be required is for all interactions with the external world to be remembered by G. Since the restarted computation will retrace its steps up to the point that G detected the problem, except now mediated by an extra level of reflective processor program, the replayed computation is guaranteed to be the same as it was the last time. The replay up until this point could therefore be performed without external world interaction—i.e., by blocking output and using remembered inputs instead). Then when it reaches this same point, interac-

retically reasonable, even though this whole approach is not especially practical.

It would be far better, of course, if there were some computationally· tractable way of inferring the instantaneous state of the level $n+1$ RPP from the instantaneous state of the level n one. This suggestion, which would mean that computations would not need to be restarted, is not as unlikely as it might first seem. The processing that goes on at adjacent levels is always strongly correlated (since, after all, level $n+1$ essentially "implements" level n). Adjacent levels are related by "meta"-ness; it is not as if different levels have "minds of their own." If it were possible to make such a step, one could refine the implementation strategy so as not to restart the computation when an impasse was reached, but rather to "manufacture" the state that would have existed one level up, had the implementation been explicitly running at that level from the beginning.

In other words, the overall strategy would be improved if the actual implementation processor could make an *instantaneous shift up*, when needed, to where it would have been had an extra level of explicit RPP been in effect since the start. Call such a modified implementation processor G′. Thus a $\Delta=n$ program would be run directly by G′ until it was discovered that $n>1$, at which time the internal state of G′ would be used to create the explicit state that would be passed to the explicit RPP that would take over running the user program. After modifying its own internal state to reflect what would have been the state one level up, G′ could devote its attention to running the RPP. This means that the original program will now be run indirectly. It will continue to be run that way until such time as it is revealed that $n>2$, at which time G′ would shift up again, and will running the base-level program double-indirectly. And so on.[12]

Over the course of the computation, in other words, G′ will gradually climb to higher and higher reflective levels. Although

tion can be resumed in a normal fashion.

[12] We are assuming (not unreasonably) that the point at which it is determined that $\Delta>l$ is a point at which all upper levels *would have been boring so far*, even if they had been run explicitly. A more formal treatment would make this explicit.

its strategy for shifting levels is not very sophisticated, G′ exemplifies the fundamentally important idea of a **level-shifting** implementation. All of the implementation processors we will discuss in the rest of the paper are level-shifting as well; they merely have more complex shifting strategies.

Invariably, each additional level of indirection will degrade the system's performance with respect to the bottom level of the user program. This is not a minor concern, given that processor overhead is typically measured in orders of magnitude. What we would really like is an implementation processor that will *never run at any higher level than necessary.* Not only should the implementation be able to **shift up easily**, in other words; it should also be able to **shift back down** whenever it discovers that things are getting boring—i.e., when it starts processing kernel expressions again.

To make this formal, we have to define local rather than global notions of boredom and introspective degree, but those are relatively straightforward extensions. That is, when it appears that the program that the implementation processor is running directly has a *local* $\Delta=0$, the implementation processor should compensate by absorbing the explicit state of the RPP it was previously running directly, and proceed to take direct responsibility for running of the computation formerly one level below. This ensures maximum utilization of the capability of the implementation processor to directly run arbitrary $\Delta=1$ computations. An actual implementation will be called **optimal** if it *never processes a kernel expression indirectly.*

There are two subtleties here. First, it may be reasonable to expect that every RPP will permit the appropriate determination of local boredom. Once the user has been able to run code at a meta level, there would seem to be no telling what might have been done there. Some sort of "time bomb" might have been· planted that will detonate at some later point in time. If, however, the local notion of boredom just cited can be used to say that a *local portion* of a program is boring, even if some of its embedding context is not, then the implementation can depend on the fact that it is safe to turn its back on an arbitrary number of boring levels of processing, just so long as it can turn around and shift back up the moment any of them becomes interesting again. In other words, it would seem in general to be very difficult to determine whether it is safe to shift down. On the other hand, as the 3-LISP example will show in some detail, there are some reasonable assumptions and techniques that enable optimality at least to be approached.

Second, we said above that, when shifting down, the implementation should **absorb** the explicit state of the RPP it was previously running directly. It takes some care to determine just what it is to absorb this state in such a way that it can later be rendered explicit, should the need arise, as the discussion of 3-LISP will show.

In broad terms, these considerations lead to an adequate implementation strategy. A **correct** implementation is one that engenders the same computation as that specified by the limit, as $n \rightarrow \infty$, of a tower of n reflective processor levels run at the top (nth) level by an actual processor. The base case for an efficient but correct processor requires an independent specification of the capabilities of an implementation processor capable of running only $\Delta=1$ programs. The induction step shows that adding an extra level of processing engenders exactly the same computation while increasing by one the maximum degree of introspection that can be handled. In order to produce a level-shifting implementation we also need computationally effective rules for determining when and how to shift up and back down.

3 3·LISP: a Reflective Dialect of LISP

Before we can make this all more precise, we need a specific reflective language to use as an example. 3-LISP[13] is a reduction-based, higher-order, lexically scoped dialect of LISP whose closest ancestor is SCHEME.[14] Other than its reflective capabilities (described below), the most significant way in which 3-LISP differs from its ancestors is that the notion of *evaluation* is rejected in favour of a rationalized semantics based on the orthogonal notions of:

[13] Smith (1982a).

[14] «Refs»

1. **Reference:** what an expression *designates, stands for, refers to, names*); and
2. **Simplification:** how an expression is handled by the 3-LISP processor; what is *returned*.

Specifically, all 3-LISP expressions are taken as designating something; the 3-LISP processor then embodies a particular form of simplification called **normalisation**, in which each expression is reduced to a *normal-form codesignator*. The motivation for and semantics of such a language are discussed in (Smith 84).

In 3-LISP, `$T` designates truth and `$F` designates falsity. Expressions of the form $[X_1\ X_2\ \ldots\ X_n]$ designate the abstract sequence of length n consisting of the objects designated by the X_i in the specified order. Expressions of the form `(F . A)` designate the value that results from applying the function designated by `F` to the argument designated by `A`. The common case of applying a function to a sequence of n (≥0) arguments $(F\ .\ [X_1\ X_2\ \ldots\ X_n])$ is abbreviated $(F\ X_1\ X_2\ \ldots\ X_n)$. The standard sequence operations are named `EMPTY`, `1ST`, `REST`, `PREP`, and `SCONS` (corresponding to LISP 1.5's `NULL`, `CAR`, `CDR`, `CONS`, and `LIST`, respectively).

As is clearly indicated for any reflective language, 3-LISP contains numerous facilities for quotation and general reference to other program structures. In general, if `X` is any expression, the quoted expression `'X` is used to designate `X` (`'X` is a primitive notation; it is not an abbreviation for `(QUOTE X)`). When one deals with quotation, one needs names for expressions of various types. We say that `'100` designates the **numeral** `100` (which in turn designates the number one hundred); `‘$T` designates the **boolean** `$T`; `‘[1 2]` designates the **rail** `[1 2]`; `'FOO` designates the **atom** `FOO`; `'(X . Y)` designates the **pair** `(X . Y)`. There are also normal form function designators called **closures,** which have no adequate printed representation. The expressions `''FOOO`, `''[1]`, and `''''$F` designate the **handles** `'FOO`, `'[1]`, and `'''$F`, respectively. The standard functions `NUMERAL`, `BOOLEAN`, `RAIL`, `ATOM`, `PAIR`, `CLOSURE`, and `HANDLE` are characteristic functions for the seven kinds of expressions just listed.

The standard operations on sequences are polymorphic, applying equally to rails. The additional standard operation `RCONS` can be used to construct new rails: `(RCONS)` designates the empty rail `[]`.

The standard operations on pairs are named PCONS, CAR, and CDR; (PCONS 'A 'B) designates the pair (A . B); (CAR '(A . B)) designates the atom A; and (CDR '(A . B)) designates the atom B. The standard operations on closures are named CCONS, ENVIRONMENT, REFLECTIVE, BODY, and PATTERN. The standard composite expression used to designate functions is of the form

```
(LAMBDA type pattern body)
```

where type is usually either SIMPLE (for non-reflective procedures) or REFLECT (for reflective procedures). Thus

```
(LAMBDA SIMPLE [N] (+ N 1))
```

designates the successor function.

Despite the many minor differences between the languages, readers familiar with SCHEME should have little difficulty understanding 3-LISP programs. The reader is referred to (Smith 84) for a more complete introduction to both the language and to the intuitions that guided its development. Very much like the metacircular interpreters discussed in the "Lambda papers,"[15] we present in figure 3 the continuation-passing 3-LISP RPP.[16]

As mentioned above, 3-LISP is based on a notion of expression reduction, rather than evaluation: the processor (NORMALISE, in place of the more standard EVAL) returns a co-designating normal-form expression for each expression it is given; see (Smith 84). We write X ⇒ Y to mean that X normalises to Y. For example:

```
                      (+ 1 2)  ⇒  3
                (PCONS 'A 'B)  ⇒  '(A . B)
((LAMBDA SIMPLE [X] (* X X)) 4)  ⇒  16
```

The code for the 3-LISP RPP is given in figure 3. All the procedures in the RPP code, other than those explicitly defined, are straightforward, side-effect-free, data manipulation functions. None have any special control responsibilities (except COND, DEFINE, and BLOCK, whose definitions have been omitted only to shorten the presenta-

[15] Sussman & Steele (1975); Steele & Sussman (1976, 1978a, 1978b, 1980); Steele (1976, 1977a, 1977b).

[16] Note: variable names ending in '!' are used, by convention, to indicate that they will always designate normal-form structures.

```
(define READ-NORMALISE-PRINT
.. (lambda simple [level env]
..... (normalise (prompt&read level) env
........ (lambda simple [result]                   ; REPLY continuation
........... (block (prompt&reply result level)
.................. (read-normalise-print level env))))))
(define NORMALISE
.. (lambda simple [exp env cont]
..... (cond [(normal exp) (cont exp)]
.......... [(atom exp) (cont (binding exp env))]
.......... [(rail exp) (normalise-rail exp env cont)]
.......... [(pair exp) (reduce (car exp) (cdr exp) env cont)])))
(define REDUCE
.. (lambda simple [proc args env cont]
..... (normalise proc env
........ (lambda simple [proc!]                    ; PROC continuation
........... (if (reflective proc!)
............... (↓(de-reflect proc!) args env cont)
............... (normalise args env
................. (lambda simple [args!]            ; ARGS continuation
..................... (if (primitive proc!)
......................... (cont ↑(↓proc! . ↓args!))
......................... (normalise (body proc!)
.................................... (bind (pattern proc!) args! (environment p
.................................... cont))))))))
(define NORMALISE-RAIL
.. (lambda simple [rail env cont]
.... (if (empty rail)
........ (cont (rcons))
........ (normalise (1st rail) env
........... (lambda simple [first!]                ; FIRST continuation
.............. (normalise-rail (rest rail) env
................. (lambda simple [rest!]           ; REST continuation
..................... (cont (prep first! rest!)))))))))
(define LAMBDA
.. (lambda reflect [[kind pattern body] env cont]
.... (cont (ccons kind ↑env pattern body)))))
(define IF
.. (lambda reflect [[premise c1 c2] env cont]
..... (normalise premise env
........ (lambda simple [premise!]                 ; IF continuation
........... (normalise (ef ↓premise! c1 c2) env cont)))))
```

Figure3 — The 3-LISP Reflective Processor Program (RPP)

tion). PROMPT&READ and PROMPT&REPLY issue the system's 'level>' and 'level=' prompts, and perform input and output, respectively, but are otherwise innocuous. ↑ and ↓[17] mediate between a structure and what it designates. Some examples:

```
  ↑(+ 2 2)  ⇒  '4
 ↑↑(+ 2 2)  ⇒  ''4
      ↓''4  ⇒  '4
↓''(+ 2 2)  ⇒  '(+ 2 2)
```

There are no hidden procedures; user programs may use CCONS (the closure constructor), BODY, NORMALISE, etc.—even ↑ and ↓ — with impunity.

By defining special reflective procedures, using

```
(LAMBDA REFLECT ...))
```

the user may augment the processor just shown. These reflective procedures are handled by line 18 of REDUCE:

```
(↓(dereflect proc!) args eny cont)
```

Thus suppose foo is bound to a reflective procedure. When the level 1 processor encounters (foo e_1 ... e_n) in the program it is running, the reflective procedure associated with the name foo is called at the *same* level as the processor, with exactly three arguments: a designator of the non-normalised argument structure '[e_1 ... e] (from the original level 0 pair), the variable binding environment, and the continuation. In this way, the user's program may gain access to all of the state information maintained by the processor that is running it. From this unique vantage point, it is easy to realize new control constructs, such as CATCH and THROW, or to implement a resident debugger.

The infinite tower appears to the user exactly as if the system had been initialized in the following manner:

```
4) (read-normalise-print 3 global)
3) (read-normalise-print 2 global)
2) (read-normalise-print 1 global)
1>
```

[17] Notational abbreviations for UP and DOWN, respectively—called NAME and REFERENT in Smith (1982).

The user can verify this by defining a QUIT procedure that returns a result instead of calling the continuation, thereby causing one level of processing to cease to exist:

```
1) (define QUIT (lambda reflect [args eny cont] 'DONE))
1= QUIT

1> (quit)        ; QUIT is run as part of the level 1 processor
2= 'DONE         ; which it kills

2> (+ 2 (quit))  ; This time QUIT terminates the level 2 processor
3= 'DONE

3> (read-normalise-print 1 global)      ; Levels can be re-created
1> (read-normalise-print 2001 global)   ; at will; level numbers
2001> (quit)                            ; are arbitrary.
1= 'DONE

1> (quit)
3= 'DONE
```

The following code defines (as a user procedure) the SCHEME escape operator CATCH:

```
(define SCHEME-CATCH
   (lambda reflect [[tag body] catch-enY catch-cont]
      (normalise
         body
         (bind tag
               ↑(lambda reflect [[answer] throw-enY throw-cont]
                    (normalise answer throw-env catch-cont))
               catch-env)
         catch-cont)))
```

For example, the following expression would return 17:

```
(let [[x 1]]
   (+ 2 (scheme-catch punt
                   (* 3 (/ 4 (if (= x 1)
                                 (punt 15)
                                 (- x 1)))))))
```

To some extent, a metacircular processor or RPP can be viewed as an account of a language (or at least of how it is processed) expressed within that language. As such, it "explains" various things about how the language is processed, but depending on the account, it can account for more or less of what is the case. In particular, it is important to realize what the above 3-LISP RPP does and does not explain.

The 3-LISP reflective processor was designed to be similar to standard Scott-Strachey continuation-based semantic accounts of λ-calculus based languages.[18] Its primary purpose is to explain the variable binding mechanisms and the flow of control in the course of error-free computations. The account intentionally does not say anything about how errors are processed, nor does it shed any light on how the field of data structures are implemented, nor on how input/output is carried out. These details are buried in the primitive procedures, and the reflective processor carefully avoids accounting for what they actually do. A different theory that did explain these aspects of the language could be written, yielding a different RPP, and a different reflective dialect—all of which would require a different implementation. But the basic architecture and strategies we employ would generalize to such other circumstances.

One of the many things that SCHEME demonstrated was that lexical scoping and the treatment of functions as first class citizens resulted in a cleaner LISP that no longer needed to quote its LAMBDA expressions. 3-LISP goes a step further by showing how to incorporate, in a semantically principled way, some of the other hallmarks of real systems, including; constructing programs on-the-fly; making explicit use of EVAL and APPLY; FEXPRS and NLAMBDAS; and implementing a debugger within a system.

4 Levels and Level-Shifting Processors

We explained in section 2 how an implementation of reflection might work; in this section we present the architecture for such an implementation in much more detail. Although we will use 3-LISP as a motivating example, our dependence on its idiosyncrasies will not be crucial; the actual code for a 3-LISP implementation is deferred until section 5.

4a Level Shifting in Conventional Implementations

Although procedurally reflective architectures are new, the idea of *level shifting* processors is not. Consider for example an implementation of LISP that supports both interpreted and compiled procedures definitions. In such a system, the non-compiled procedures

[18] E.g., Stoy (1977), Muchnick (1980).

will be defined by LISP source code (typically, LAMBDA expressions represented as list structure), while the compiled ones will be represented by blocks of instructions acceptable to the machine on which the LISP system is implemented. Both kinds of procedures are represented as code, but in different languages: the uncompiled source code, which will be run by the implementation, is in LISP, whereas the compiled code, which will be run by the same processor that runs the implementation (probably the CPU of the underlying machine—i.e., in machine language).

Given procedures in these two different languages, there are complexities in having them interact properly—complexities that the whole system usually smoothes over so well that the user may never be aware of them. Consider in particular the procedure-call mechanism, where some procedure A calls another procedure B. **In** the simplest case, where both A and B are represented by compiled code, the linkage is usually achieved directly using a machine language branch instruction to transfer control from A to the first instruction of B (after arguments and the return address are loaded into registers or pushed on a stack). On the other hand, when a compiled procedure A calls a B that has no compiled code associated with it, a machine-language transfer of control must be made not from A to B, but from A to the block of machine language code that implements the explicit LISP processor (EVAL) that in turn can examine the list-encoded LAMBDA expression representation of B.

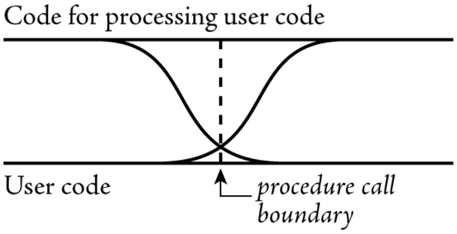


Figure 4 — Level shifting caused by calls between compiled and non-compiled procedures

Once the LISP processor is in control, the situation is reversed. As long as neither A nor B is compiled, everything is straightforward; the locus of control at the machine language level remains within the LISP processor's code, and that processor implements an appropriate connection between the LISP code for A and the LISP code for B. When a non-compiled A calls a compiled B, however, there will have to be a machine-language level transfer of control from the code for the LISP processor to the code representing B.

As depicted in figure 4, this can be described as simple level shifting between a level of *direct* processing (at the lower level, where user code is run) and one of *indirect* processing (at the upper level, where processors for user code are run). Shifting up and down both occur at times corresponding to procedure-to-procedure calls (and returns). What controls the level-shifting in this particular case is not the occurrence of reflective procedures, but rather changes in language.

In particular, we are assuming that all user code is at the lower level—i.e., that all user code is run at level 0. Some of that code is in LISP; some is in machine language. At level 1 there is a program, written in machine language, that is a processor program for LISP; call this program M_L. In this simple model, this is only one of four possible processor programs one could have; the other three being a LISP program to process machine language (L_M); a machine language program to process machine language (M_M), and a LISP program to process LISP (L_L)—i.e., a metacircular interpreter for LISP in LISP. The level shifting strategy adopted by the implementation is one that enables the implementation to get away with just (i) the one processor program M_L, and (ii) a simple underlying processor G that knows only how to run machine language programs. If it adopted a different level-shifting strategy, it might need some of those other processor programs. For example, if the implementation were not to shift down when it encountered a non-compiled A to compiled B procedure call, it would need M_M—a machine language program to interpret machine language. Similarly, if it were to try to shift up on a non-compiled to non-compiled procedure call, it would need L_L.

The analogy between standard implementations and implementations of reflection can be pushed even further by considering how matters are complicated when explicit calls to `EVAL` are supported. Suppose that the expression `(EVAL '(FOO 10))` is found within the body of a (non-compiled) procedure named `FEE`. When the implementation (specifically, the CPU running the program M_L) encounters this expression while processing a call to `FEE`, control within the user's program must pass to the `EVAL` procedure, which, we will assume for the moment, will be defined via LISP source code (i.e., we will assume that `EVAL` is bound to L_L, the metacircular processor program for LISP). The net effect will be that M_L will process the code for `FOO` indirectly specifically—M_L

will process L_L (the code for `EVAL`), which in turn will process `FOO`. So `G` (the CPU) will be two levels away from the code for `FOO`.

It is a relatively simple change to the LISP processor program M_L to have it recognize calls to `EVAL` and treat them in a special way that avoids this extra level of indirect processing—in fact that is what most implementations of LISP do (see figure 5). This change also means that the code L_L need not be kept in the system. Notice, however, that this change is another form of level shift, not between compiled code and the LISP processor this time, but between the following two different LISP expressions:

```
(EVAL '(FOO 10))   and   (FOO 10)
```

It is no coincidence that there are strong similarities between these two forms of level shifting—compiled vs. interpreted, on the one hand, and ordinary expressions vs. arguments to `EVAL`, on the other. The machine code for the LISP processor and the compiled code for `EVAL` are *exactly the same thing*: they are both M_L—a program, written in machine language, to process LISP. The downward shift to avoid an extra level of explicit processing on calls to `EVAL` is *also* the downward shift to run the compiled code for `EVAL`. **In** both cases, the relationship between adjacent levels is the same: the computation that happens implicitly at one level is being carried out explicitly one level above it.

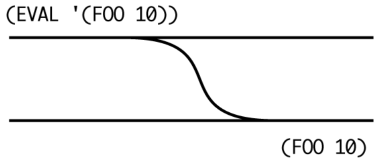


Figure 5 — Level shifting caused by calls to `EVAL`

4b Analysing a Processing Activity

While the simple level shifting techniques described above might suffice to handle a non-reflective language with explicit access to its processor, the task of implementing 3-LISP has an additional complexity; viz., reflective procedures give the user a way of running procedures at arbitrary levels of the program's processor, including programs that are themselves reflective. **In** effect, the user can get access into the middle of `NORMALISE` (3-LISP 's counterpart to `EVAL`), making the job of "compiling" `NORMALISE` more difficult than it would otherwise be. Moreover, if you look carefully at the definition of 3-LISP and at its RPP, several of the standard control constructs, such as `LAMBDA` and `IF`, look dangerously circular, since they are both defined as reflective procedures and also used in the account of how the processor works. In order to implement a generalised level-shifting processor of the sort suggested in the last section, therefore, we first have to analyse the processing activities that must go on with an eye to implementing some of them directly, while allowing others to be carried out in virtue of one or more levels of explicit processing.

In particular, we need to name various relationships between the code in a processor program and the code that such a program processes.

First, if an expression or procedure to be applied is *primitive*, or, more generally, if within the processor there is code that corresponds exactly to the expression or procedure in question, then that expression or procedure can be dealt with directly in what amounts to a single processing step. We will call such expressions and procedures **directly implemented.** Small integer arithmetic, for example, is typically directly implemented in LISP implementations by the arithmetic capabilities of the underlying machine language; primitive data structure operations (like CAR and CONS), at least in simple implementations, are also directly implemented by special procedures.

Second, if an expression is not directly implemented, it can usually be broken down into a series of constituent steps that are either themselves directly implemented, or can be broken down in turn, leading in the end to a long series of directly implemented expressions. Suppose for example we have the following definition of the 3-LISP procedure 2ND:

```
(define 2ND
   (lambda simple [x]
      (1st (rest x))))
```

Then the processing of (2ND [10 20]) can be broken down into roughly the series of simpler processing activities corresponding to the processing of (REST [10 20]) and (1ST [20]). We will call this kind of processing decomposition engendered by the standard compositional and recursive nature of programs a **horizontal decomposition,** to correspond to the way we have been depicting levels of processing. In procedure-based languages, procedure call boundaries usually serve as the most convenient dividing lines or “click points” separating these processing units. In general, a lengthy computation is carried out in virtue of its horizontal decomposition into a series of simple steps, each of which is directly implemented. (Horizontal decomposition corresponds to the standard notion of a computation tree, based on a compositional expression, with the directly implemented steps as the leaves.)

As we have seen, the existence of a metacircular processor program provides a third possible way of processing an expression. **In** particular, for any expression `X`, instead of processing `X` we can do an *upwards vertical conversion,* and process instead an expression that *explicitly represents the processing of* `X`. For example, we can convert `(2ND [10 20])` into `(NORMALISE '(2ND [10 20]) … )`. This upwards vertical conversion can then in turn be horizontally decomposed, typically into more steps than the original expression would have been decomposed into. For example, the horizontal decomposition of

```
(NORMALISE '(2ND [10 20])))
```

through `NORMALISE` and `REDUCE`, begins (roughly):

```
01:   (COND [(NORMAL ‘(2ND [10 20])) ... ]
             ... )

02:   (NORMAL '(2ND [10 20]))
03:          ...                various internal steps within NORMAL
04:   (ATOM '(2ND [10 20]))
05:   (RAIL '(2ND [10 20]))
06:   (PAIR '(2ND [10 20]))
07:   (REDUCE  (CAR '(2ND [10 20]))
               (CDR '(2ND [10 20]))
               ENV
               CONT)
08:   (CAR '(2ND [10 20]))
09:   (CDR '(2ND [10 20]))
10:   (NORMALISE '2ND
11:   (NORMAL '2ND)
12:          ...                various internal steps within NORMAL
13:   (ATOM '2ND)
14:   (BINDING '2ND ... )
15:         ...
```

Some expressions, like `(NORMALISE '3 … )`, can be converted down (to `3`, in this case), although downwards conversion is not always possible.

In sum, there are three ways in which an implementing processor can attempt to perform any given processing activity:

1. It can implement it directly;
2. It can perform a horizontal decomposition, and process the smaller steps; or
3. It can perform an upwards or downwards vertical conversion, and then process the result at a different level.

Given this flexibility, we can make the following observations concerning 3-LISP ’s various kinds of procedures:

1. Primitive procedures, such as 1ST and ↑ (UP), cannot be decomposed horizontally. Moreover, as line 18 the metacircular processor shows:

   ```
   (CONT ↑(↓PROC! . ↓ARGS!))
   ```

   and as common sense would suggest, every primitive is used in the horizontal decomposition of every (upwards) vertical conversion of it. Hence the primitives must be performed directly, or else be a part of some larger activity that is performed directly.
2. Other simple (non-reflective) procedures can be decomposed horizontally using the closure associated with the procedure. However, simple procedures that are part of the standard system and whose processing can be completely decomposed *a priori* (this certainly includes but is not limited to the kernel procedures) are also candidates for being implemented directly; e.g., 3-LISP’s BINDING and BIND.
3. Reflective procedure require one level of vertical conversion (in some sense that is what reflective procedures *are*), after which the (corresponding “de-reflected”) procedure can be decomposed horizontally using the corresponding simple closure.

4c Tiling Diagrams

The notions of horizontal decomposition and vertical conversion suggest an analogy. Imagine a simple **tiling game**, where the objective is to find a continuous path from left to right across an infinitely tall board consisting of rows of non-overlapping numbered tiles. You are only allowed to step on tiles with certain numbers, and you are never allowed to "retreat" (i.e., to move to the left). As illustrated by the simple example in figure 6, each row typically consists of more tiles than the row below. The best score is achieved by using the fewest steps, so the general strategy is to stay as low as possible on the board. On the other hand, there are two pitfalls that must be avoided: (i) you do not want to end in a dead-end (no further steps possible, necessitating a retreat, which is illegal); and (ii) you do not want to encounter a situation where you are climbing a spike without a top.

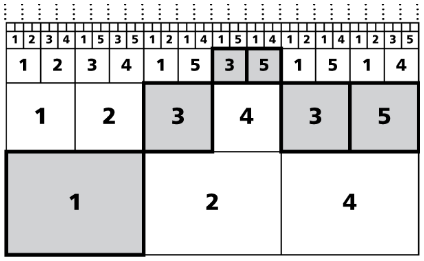


Figure 6 — Tiling Game

The board shown in figure 6 was constructed according to the following two rules:

1. Above every tile numbered *x* is a sequence of tiles y_i (listed in the form $\{x\text{: } y_\mathrm{i}\}$):

   {1: 1,2} {2: 3,4} {3: 1,5} {4: 3,5} {5: 1,4}

2. In constructing a path across the board, only odd-numbered tiles may be stepped on.

Given these rules, the best successful path is illustrated by tiles outlined with heavy lines.

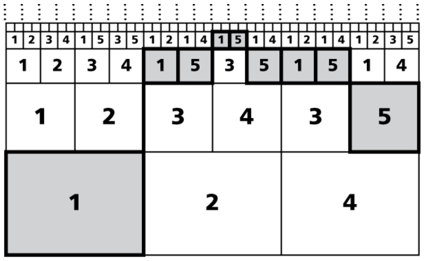


Figure 7 — Tiling game (no steps on 3-tiles)

In this example, given the particular way each tile is related to the tiles above it, it is always possible to find a path, no matter what the bottom layer of tiles is chosen to be. Moreover, it can be shown that no path ever need go higher than three rows from the bottom (in order to get over a 2-tile), and that the local strategy of choosing the lowest possible path will always be optimal and will never lead to a dead end.

If the rules were made more restrictive by forbidding you to step on 3-tiles, however, the game would still be winnable; an optimal path under these conditions is illustrated in figure 7. However, the same cannot be said of either the 1-tile or the 5-tile, both of which are unavoidable (note the insurmountable "spikes" of 1-tiles, indicated in figure 8).

To implement a reflective language is basically to play a tiling game, where:

1. Tiles correspond roughly to procedure calls;
2. Tiles above another tile are approximately (the horizontal decomposition of) an upwards vertical conversion of the lower tile;
3. Horizontal tiles represent horizontal decompositions; and
4. Tiles that can be stepped on are procedures that have a direct implementation.

Like the designer of a tiling game that admits a winning strategy, there is a twofold challenge: (i) you must carefully select a collection of processing activities that will be implemented directly (corresponding to tiles that can be stepped on); and (ii) for efficiency, you must play the game well, which means coming up with a near-optimal strategy for achieving any Δ=*n* (*n* finite) computation that, by shifting either up or down, avoids spikes and dead ends and crosses the board in a minimum number of steps.

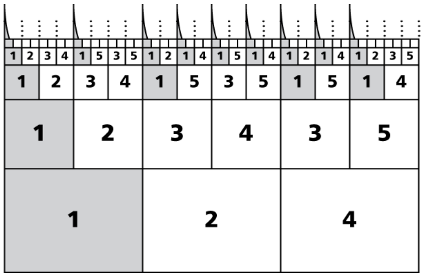


Figure 8 — Spikes of 1-tiles

4d Direct Implementation of Kernel Procedures

We said earlier that the **kernel** of a reflective language consists of those parts of the RPP that are used in the course of processing the RPP one level below. For 3-LISP, call the six procedures NORMALISE, REDUCE, NORMALISE-RAIL, LAMBDA, IF, and READ-NORMALISE-PRINT the **primary processor procedures** (PPPs), and call their embedded continuations (the REPLY, PROC, ARGS, FIRST, REST, and IF continuations identified on lines 4, 16, 20, 31, 33, and 41 of the RPP) the **primary processor continuations** (PPCs). The 3-LISP kernel then consists of:

1. The PPPs;
2. The PPCs;
3. The utilities like BINDING, BIND, and NORMAL; and
4. The primitives such as CAR, CDR, ↑, ↓ and RCONS.

If the implementation directly implemented (i.e., had "compiled" versions of) all the kernel procedures, it would be guaranteed that any Δ=*n* (*n* finite) expression could be normalised (the analogous situation in the tiling game would be one where any tile on rows *n* and above could be stepped on). The tiling analogy makes it clear why it is the *kernel* procedures, not the *primitive* procedures, for which we need direct implementations: since all primitives are used in the horizontal decomposition of every vertical conversion of them, primitives will form spikes in the tiling diagram, over which no shifting strategy will be able to climb.

As we will discuss later, an implementation can be slightly more minimal (directly implement fewer procedures), but directly implementing the whole kernel makes for the simplest processor code, and the simplest shifting strategies. As with the tiling game, the choice of a basis set cannot be made independently of the strategy for shifting up and down.

4e When and How to Shift Up

The next important problem is to determine (i) the criteria by which the implementation processor will decide that it is necessary to shift up, and (ii) the mechanisms for achieving this transition. We begin by observing that the state explicitly maintained at each level of processing by the reflective processor consists of the expressions, environments, and continuations that are passed as arguments among the PPPs. Not captured at any particular level are the global state of input/output streams and the structural field itself; fortunately, however, the RPP does not use side effects to remember state information (except when the program that it is running forces it to *process* a side effect).[19] As a result, when a shift up oc-

[19] Although 3-LISP has primitive procedures that "smash" structures, in this paper we will pretend that there are not any. Without this simplifying assumption, bothersome technicalities would tend to obscure the otherwise

curs, only an expression, an environment, and a continuation will have to be "pulled out of thin air."

Shifting up will have to occur when control would leave the implementation code that represents the directly implemented kernel. This can happen at only a handful of places in the RPP: at one of the continuation calls, (cont …), and on line 18, where reflective procedures are called using the expression:

```
(↓(de-reflect proc!) args env cont)
```

The real question is where in the implementation processor should the shift up take us? In other words, it is one thing to know where one needs to leave the level below and shift up; it is much less clear where, in the level above, one should *arrive*.

Four possibilities suggest themselves. First, it would seem that the implementation processor could shift from processing (cont exp) to processing the following upwards vertical conversions of (cont exp):

```
(normalise '(cont exp) e? c?)
```

Second, on the other hand, inspection of the RPP shows that this is equivalent to:

```
(reduce 'cont '[exp] e? c?)
```

And if we assume that exp and cont normalise to exp! and the simple (non-reflective) closure cont!, respectively, both of these are equivalent to:

```
(reduce ↑cont! '[exp] e? c?)
(reduce ↑cont! ↑[exp!] e? c?)
```

Since the higher level will in general be finer-grained (go through more identifiable steps) than the level below it, there is not a definitive choice to made among these. Given our particular choice of PPPs, all four of these possibilities are acceptable. Pure efficiency would suggest the last, since it is the "furthest along" in the processing. This in turn suggests an even more efficient answer, and a more natural seam, at line 23 in the ARGS continuation at the instant NORMALISE is about to be called on the body of the (simple)

straightforward solution. The interested reader is referred to the *Interim 3-LISP Reference Manual* (Smith & des Rivieres 1984) which contains a correct implementation for the unabridged language.

cont! closure:

```
(normalise (body ↑cont!)
           (bind (pattern ↑cont!)
                 ↑[exp!]
                 (environment ↑cont!))
           c?)
```

Since exp! and cont! are part of the state of the implementation, and since this expression does not use an environment, only the continuation *c?* needs to be pulled out of thin air. What should this continuation be? The (somewhat surprising) answer is that the appropriate continuation is *not* a function of the current level of processing; rather, it is a function only of the last processing done at the next higher level!

Why is this the case? The real answer is that it is because 3-LISP 's RPP can be processed directly by a finite state machine, but it is important to see why this is so. There are two critical things to realise.

First, the RPP implements a "tail-recursive" dialect of LISP (e.g., SCHEME;[20] it is not procedure calls *per se* that cause the processor to accumulate state, but rather only *embedded* procedure calls. For example, with respect to a call to the procedure represented by (lambda simple [x] (f (g x))), the call (g x) is embedded in the first argument position of (f (g x)), and therefore requires the processor to save state until (g x) returns, just as in a conventional implementation of procedure calls. The call to f, on the other hand, is not embedded with respect to the initial call (rather, it substitutes for it), and can be implemented much like a GO-TO statement, except that arguments must be passed as well. The fact that 3-LISP has a tail-recursive processor can be seen by inspecting the RPP and observing that

1. The number of bindings in an environment is a (more-or-less) linear function of the static nesting depth of programs; and

2. When a call to a simple procedure is reduced, the continuation in effect upon entry to REDUCE is the one passed to NORMALISE for the body of the called procedure's closure.

[20] Steele & Sussman (1976a).

The key implication of this is that when one procedure calls another from a non-embedded context, the continuation carried by the processor upon entry to the called procedure is the same as what it was upon entry to the calling procedure.

The second crucial property is that the PPPs always call one another in non-embedded ways. Together with the first observation, this implies the following property of the reflective processor processing the RPP itself:

> *The continuation carried by the processor upon entry to any* PPP is *always the same.*

This assertion can be phrased more precisely:

> *The (level 2) reflective processor (*RPP*) processing the (level 1)* RPP *processing a (level 0) Δ≤1 structure always carries the same level 2 continuation at every trip through level 2* `REDUCE` *when the level 2* `PROC` *is bound to* `'NORMALISE`.

In other words, if one were to "watch" the level 2 state upon entry to `REDUCE`, one would find that `CONT` was always bound to the same closure whenever `PROC` is bound to the atom `'NORMALISE` (or `'REDDUCE`, or `'CONT`, etc.).

Since the points in the RPP where the shift up will happen correspond to non-embedded calls within it—specifically, either to `(↓(de-reflect proc!) args env cont)` or to one of the six `(cont … )` expressions—the continuation that must be reified is *not* a function of the current level of processing. Instead, it is the last continuation that was explicitly used at that level, which will be the *original* `REPLY` continuation at the next higher level, if user-defined code has never been run at that level before.

4g When and How to Shift Down

Deciding when to shift down is similarly straightforward. The implementation processor should shift down whenever it is asked to process something that is directly implemented. In practice, it is not necessary to shift down as soon as possible (i.e., full optimality need not be achieved); it suffices to recognize only the situation where the implementation processor is processing calls to PPPs and PPCs, since all paths through the RPP will pass through these procedures. The situation can be detected in the code corresponding to the ARGS continuation (i.e., is PROC! bound to the closure for a PPP or PPC?). It is also essential that the arguments passed to the PPPs be scrutinized, to ensure that they are "reasonable" (of proper type and so forth). If they are, the implementation processor can perform a downwards conversion from (for example):

```
(normalise (body ↑normalise)
           (bind (pattern ↑normalise)
                 args!
                 (environment ↑normalise))
           cont)
```

to

```
(normalise (1st ↑args!)
           (2nd ↑args!)
           (3rd ↑args!))
```

The continuation in effect prior to shifting down must be recorded in the absorbed state. Typically, it will be a REPLY continuation—the original one for that level of processing, born within the call to READ-NORMALISE-PRINT that created that level at the time of system genesis. However, since it is possible for the user to write code that calls NORMALISE from an embedded context, it is essential to save the continuation each time a downward shift occurs so that it may be brought back into play the next time the processor shifts up to this level.

How is it that we can store away a user-supplied continuation and shift down, without knowing what behaviour that continuation will engender? The answer is simply that that continuation will not be called—cannot come into play—until such time as the computation at the lower level returns a result. Since each PPP ends in a tail-recursive call, this chain can break down only if some *non*-PPP is called which returns a result instead of calling the continuation passed to it. But it is precisely these calls that always cause a shift up (see the definition of &&CALL in the next section); hence, the implementation processor will automatically find its way back to the appropriate level whenever a non-primary processor continuation would be called at a higher level.

5 A 3-LISP Implementation Processor Program

The principal reason that the 3-LISP RPP cannot serve as a model for a real implementation (i.e., cannot be translated directly into an appropriate implementation language like machine language or C) is that it is not a closed program. As indicated in line 18 of the RPP, the processing of reflective procedures causes the locus of control to leave the PPPs and venture off into code supplied by the user. In the last section we gave a general description of how to write a real implementation that avoided this problem; in this section we use those strategies and present a full closed program for a real implementation of 3-LISP. This program will be expressed in a conservative subset of 3-LISP; no crucial use will be made of 3-LISP's meta-structural, reflective, or higher-order function capabilities. We have chosen to write this real implementation of 3-LISP in 3-LISP (i.e., to write a true metacircular processor for 3-LISP) because it allows us to suppress many implementation details that would necessarily surface if a different language were chosen. The most important omissions are the memory representation of the elements of the structural field, garbage collection, error detection and handling, and all input/output. While important, these concerns, which 3-LISP shares with other LISP dialects, are not germane to our particular topic of how to implement procedural reflection. What this program will do is to discharge all of the salient issues having to do with reflection; translating from the code presented here to an implementation in a more reasonable implementation language would be straightforward.

5a The Basic Implementation Processor

As noted in earlier sections, the structure of the 3-LISP implementation processor program will be based on the structure of the RPP itself. Specifically, for each PPP there is a corresponding implementation processor procedure bearing its source's name prefixed by '&&'; e.g., &&NORMALISE implements NORMALISE, As will be discussed later, each takes an additional parameter named STATE that represents the absorbed state, which is used only when shifting up or down (such shifts will be indicated with underlined code). The following is the code for the implementations of NORMALISE and REDUCE (&&NORMALISE-RAIL and &&READ-NORMALISE-PRINT, derived in an analogous manner, are given in the appendix):

```
(define &&NORMALISE
   (lambda simple [state exp env cont]
      (cond [(normal exp) (&&call state cont exp)]
             [(atom exp) (&&call state cont (binding exp env))]
             [(rail exp) (&&normalise-rail state exp env cont)]
             [(pair exp)
              (&&reduce state (car exp) (cdr exp) env cont)])))

(define &&REDUCE
   (lambda simple [state proc args env cont]
      (&&normalise state proc env
         (make-proc-continuation proc args env cont))))
```

Similarly, for each type of PPC there is a corresponding implementation processor procedure with names of the form &&xxx-CONTINUATION. E.g., &&PROC-CONTINUATION implements the "PROC" type continuations (see lines 16–25 of the RPP), which field the result of normalising the procedure part of a pair. While the RPP continuations are closed in an environment in which a handful of non-global variables are bound, their implementation equivalents are passed these data as explicit arguments (e.g., &&PROC-CONTINUATION is passed as arguments the bindings of PROC, ARGS, ENV, and CONT from the incarnation of &&REDUCE that spawned it). &&EXPAND-CLOSURE (presented below) implements the last clause of the "ARGS" continuation, although it does not correspond to a continuation on its own. Again, two examples (the others are given in the appendix):

```
(define &&PROC-CONTINUATION
   (lambda simple [state proc! proc args env cont]
      (if (reflective proc!)
           (&&call state ↑(de-reflect proc!) args env cont)
           (&&normalise state args env
              (make-args-continuation proc! proc args env cont)))))

(define &&ARGS-CONTINUATION
   (lambda simple [state args! proc! proc args env cont]
      (if (directly-implemented proc!)
          (&&call state cont ↑(↓proc! . ↓args!))[x]
          (&&expand-closure state proc! args! cont))))
```

Note that &&ARGS-CONTINUATION simply executes any procedures which are implemented directly, using the same technique that is used in the RPP for primitives. If this code were to be translated

[x] In the published paper this line was erroneously printed as
(&&call state cont ↑(↑proc! . ↑args!))

into a different implementation language, the ↑(↓proc! . ↓args!) expression would be turned into appropriate calls, for each directly implemented procedure, to the procedure that performs the direct implementation.

As well as defining these implementation procedures to do the work of the ppcs, the implementation must also contain code to create instances of the processor continuations exactly as specified by the RPP —i.e., it must create the exact PPC closures that would have been created had the RPP been used explicitly. Such continuations will never be used by the implementation as such, but since they are visible from user code they must be perfectly simulated.

There are four procedures in the implementation to construct closures of each of the four types. For example, the

```
(make-proc-continuation proc args env cont)
```

expression in &&REDUCE will produce the same closure that lines 16-25 in REDUCE would, given identical bindings for the four variables. An example (the others are given in the appendix):

```
(define MAKE-PROC-CONTINUATION
   (lambda simple [proc args env cont]
      ↑(ccons 'simple ↑(bind '[proc args env cont reduce]
                             ↑[proc args env cont reduce]
                             global)
         '[proc!]
         '(if (reflective proc!)
               (↑(de-reflect proc!) args env cont)
               (normalise args env
                  (lambda [args!]
                     (if (primitive proc!)
                         (cont ↑(↓proc! . ↓args!))
                         (normalise (body proc!)
                                    (bind (pattern proc!)
                                          args!
                                          (environment proc!))
                                    cont))))))))
```

In many cases the implementation procedures call one another, in exactly those places where the PPPs in the RPP call other PPPs. For example, &&NORMALISE calls &&REDUCE in just the place (line 12) where NORMALISE would call REDUCE. However, in those cases where it is not possible to determine *exactly* which procedure to call, the implementation procedures defer this task to &&CALL. E.g., whereas in lines 9 and 10 of the RPP NORMALISE calls the procedure designated by the local variable CONT, the corresponding lines in &&NORMALISE pass the buck to &&CALL, which inspects the closure designating the function to be called. If the closure is a PPP or a PPC, the corresponding implementation procedure (&&...) is invoked. In the case of PPCs, the non-global bindings captured within them must be extracted and passed as extra arguments to the implementation versions, as discussed earlier. (The two shift-up cases will be discussed below.)

```
(define &&CALL
  (lambda simple x
    (let [[state (1st x)] [f (2nd x)] [a (rest (rest x))]]
      (cond [(ppp ↑f) (&&call-ppp state fa)]
            [(ppc ↑f) (&&call-ppc state f (1st a))]
            [(directly-implemented ↑f)
             (&&call (shift-up state)
                     (reify-continuation state)
                     ↑(f . a))]
            [$t (&&expand-closure (shift-up state)
                    ↑f ↑a (reify-continuation state))]))))

(define &&CALL-PPP
  (lambda simple [state f a]
    ((select (ppp-type ↑f)
        ['normalise &&normalise]
        ['normalise-rail &&normalise-rail]
        ['reduce &&reduce]
        ['read-normalise-print &&read-normalise-print]
        ['if &&if]
        ['lambda &&lambda])
     . (prep state a))))

(define &&CALL-PPC
  (lambda simple [state f arg]
    (select (ppc-type ↑f)
        ['proc  (&&proc-continuation state arg (ex 'proc f)
                    (ex 'args f) (ex 'env f) (ex 'cont f))]
        ['args  (&&args-continuation state arg (ex 'proc! f)
                    (ex 'proc f) (ex 'args f) (ex 'env f)
                    (ex 'cont f))]
        ['first (&&first-continuation state arg (ex 'rail f)
                    (ex 'env f) (ex 'cont f))]
        ['rest  (&&rest-continuation state arg (ex 'first! f)
                    (ex 'rail f) (ex 'env f) (ex 'cont f))]
        ['reply (&&reply-continuation state arg (ex 'level f)
                    (ex 'env f))]
        ['if    (&&if-continuation state arg (ex 'premise f)
                    (ex 'c1 f) (ex 'c2 f) (ex 'env f)
                    (ex 'cont f))])))
```

5b Shifting Up, Shifting Down, & Level Management

The implementation presented so far will correctly process code at a given level; we need next to examine shifting back and forth between levels. This will enable us to explain the underlined clauses in the definition of &&CALL, above.

If an expression with Δ>1 is given to &&NORMALISE, then at some point a pair involving a user-defined reflective procedure will be given to &&REDUCE. This in turn will go to &&PROC-CONTINUATION, will pass the test for reflective closures, and will generate a call to &&CALL with a (corresponding de-reflected) closure that &&CALL fails to recognise as one for which there is an implementation equivalent. The last (underlined) COND clause in &&CALL handles this case, while ensuring that the locus of control remains within the code of the implementation processor program. As discussed earlier, the implementation processor must shift up, altering its internal state to accurately reflect what would have been happening at the next higher processing level in the tower.

In order to understand this clause, imagine that instead it was replaced with the single clause [$t (f . a)]. In some sense this would "work" (since we are writing the implementation processor in 3-LISP), but it would violate our goal of making the implementation be a closed program. The procedure f is intended to be called at this level, but we cannot afford to use it in the implementation, because we did not write it and therefore do not know that it stays within the restricted subset of 3-LISP that the implementation is allowed to use. If, for example, it contained reflective code, that would cause the *implementation* processor to reflect, whereas what we want is for the implementation processor to *model* (i.e., *implement*) that reflection. So instead of using the (f . a) clause, the implementation processor must instead shift up, effectively converting (f . a) into (REDUCE ↑f ↑a … …). By assumption, we know that f is bound to a non-reflective, non-primitive closure, which means we will want to decompose it horizontally, so this call to REDUCE is equivalent to (&&EXPAND-CLOSURE … ↑f ↑a …). To make this work we need to supply two missing arguments: a continuation for the next higher level of processing (the second '…'), and a new STATE argument for all levels above that (the first '…'). As discussed in section 4, the continuation can simply be taken from the top of the absorbed state stack, which is done by REIFY-

CONTINUATION. SHIFT-UP then returns the (saved) states for all levels above that.

If, on the other hand, f is primitive, kernel, or some other procedure that we have directly implemented, we can simply use (f . a). This is the case handled by the third (first underlined) clause in &&CALL. Performing the procedure application is not difficult (effected with ↑(f . a)); the question to be asked is what to do with the result that is immediately returned. The answer is that it needs to be sent to that continuation that is waiting for a result from this level of processing. We can find that continuation at the top of the absorbed state stack, which might make us think we could simply do ((shift-up state) ↑(f . a)). But that would be to assume that we also have a direct implementation for that continuation, which will not necessarily be true.[x] So we first do the (f . a), and then immediately shift up and recursively ask &&CALL to figure out how to give the result to the appropriate saved continuation.

Note that this last case is one where the processor is asked to *use* a primitive or kernel procedure, not one where it is asked to *process* a primitive or kernel procedure, a situation which is dealt with straightforwardly in the fourth line of the definition of &&ARGS-CONTINUATION.

The corresponding shift down operation can occur whenever the implementation processor finds itself processing a structure that it knows how to process directly, which will include directly implemented procedures, PPPs, and PPCs. Since the locus of control must stay within the "&&" procedures, &&EXPAND-CLOSURE, when it detects that the closure it is about to expand is of such a type, can shift down and call the corresponding implementation processor procedure directly. This would suggest the following code:

[x] As mentioned in the "2010 Perspective" at the beginning, this is the issue that we recognized that the implementation in (Smith 1984) did not handle properly.

```
;;; (define &&EXPAND-CLOSURE
;;;    (lambda simple [state proc! args! cont]
;;;       (if (or (directly-implemented proc!)
;;;               (ppp proc!)
;;;               (ppc proc!))
;;;           (&&call (shift-down cont state) ↑proc! ↑args!)
;;;           (&&normalise state
;;;                        (body proc!)
;;;                        (bind (pattern proc!)
;;;                              args!
;;;                              (environment proc!))
;;;                         cont))))
```

However there are two problems with this definition. First, &&EXPAND-CLOSURE will never be called with a directly implemented procedure, since &&ARGS-CONTINUATION and &&CALL check for that case before calling &&EXPAND-CLOSURE. This is reasonable, because even though in some sense we *could* shift down, as explained above we would immediately have to shift back up again, in order to figure out what to do with the result. So only the PPPs and PPCs are relevant. We cannot blindly shift down upon encountering them, because our implementation versions make rather strong assumptions about the arguments they are given, and we therefore need to check that the arguments we are given explicitly conform to these assumptions. Note for example that reflective continuations are well-formed—i.e.:

```
(NORMALISE 'x global (lambda reflect [a e c] (c ↑a)))
```

normalises to

```
'[(binding exp env)]
```

However our implementation versions assume that continuations are simple closures that normalise their arguments. Since there is no conceptual problem with *not* shifting down—all it means is that processing will be one level more indirect than may be strictly necessary—we adopt a version of &&EXPAND-CLOSURE that checks these integrity conditions, and shifts down only if they are met. Furthermore, we shift down only on NORMALISE and the PPCs; the other PPPs could be checked, but that would only add complexity (idiosyncratic argument integrity checks), and, as an inspection of the RPP shows, there will only be one extra horizontal processing step before a call to NORMALISE is encountered, so this will not be a very serious inefficiency.

All of these considerations lead us to the following definition. SHIFT-DOWN is used to absorb the continuation into the absorbed states of the higher levels.

```
(define &&EXPANO-CLOSURE
   (lambda simple [state proc! args! cont]
      (cond [(and (= (ppp-type proc!) 'normalise)
                  (plausible-arguments-to-normalise args!))
             (&&normalise (shift-down cont state)
                 ↑(1st args!) ↑(2nd args!) ↑(3rd args!))]
            [(and (ppc proc!)
                  (plausible-arguments-to-a-continuation args!))
             (&&call-ppc (shift-down cont state)
                        ↑proc!
                        ↑(1st args!))]
            [$t (&&normalise state
                   (body proc!)
                   (bind (pattern proc!)
                         args!
                         (environment proc!))
                   cont)]])))
```

The only further issue having to do with level shifting is determining the structure of the continuations saved for each level of the infinite tower. The initialization process described in section 3 would result in one REPLY continuation per level as the initial conditions. Since we naturally defer the creation of the level *n* initial continuation until such time as the implementation processor needs to reify it, the absorbed state of the whole tower can in fact be represented as a (finite) sequence of continuations for the intervening levels from the current level of the implementation processor up to the highest level reached to date. There is one subtlety; since each CREPLY continuation is closed in an environment in which level is bound to the integer level number, we store as the last element of this continuation sequence the level number for the next level not yet reached. The implementation processor is started off at level 1 in the code corresponding to READ-NORMALISE-PRINT; hence the initial absorbed state, which represents a (virtual) tower of initial continuations for levels 2 to ∞, consists of the singleton sequence [2].

```
(define 3-LISP
   (lambda simple []
      (&&read-normalise-print (initial-tower 2) 1 global)))
```

```
(define INITIAL-TOWER
   (lambda simple [level] (scons level))

(define SHIFT-DOWN
   (lambda simple [continuation state]
      (prep continuation state)))

(define REIFY-CONTINUATION
   (lambda simple [state]
      (if (= (length state) 1)
          (make-reply-continuation (1st state) global)
          (1st state))))

(define SHIFT-UP
   (lambda simple [state]
      (if (= (length state) 1)
          (scons (1+ (1st state)))
          (rest state))))
```

5c Summary

As was discussed in section 4, as long as the set of implemented procedures is broad enough to ensure that every call to a kernel procedure will "top out" at some finite level, there is no need for the implementation processor to handle *every* kernel utility procedure (e.g., NORMAL and BIND). In the code just presented we have included the appropriate code to handle these kernel utilities as if they were primitive procedures, but some of them need not have been so included. Though there is probably no unique solution, there are no doubt more "minimal" implementations, in the sense of implementations that directly implement fewer 3-LISP procedures; it is a bit of an exercise to figure out exactly how few are minimally necessary. In a real implementation, however, efficiency presses the other direction, towards implementations that implement *more* utilities—a requirement that can usually be met, provided they do not involve non-standard control constructs, and are not "open" in the sense of calling user-supplied arguments as procedures (i.e., are not higher-order).

Given the code we have presented, it is easy to verify by inspection that all "&&…" procedures are used in the following restrictive ways:

1. They are always called from other "&&…" procedures, with the exception of 3-LISP which is the root procedure;
2. They are always called from non-embedded contexts;
3. They never use, either directly or indirectly, any reflective

procedure other that those for the standard control structures;

4. They are never passed as an argument, or returned as a result;
5. They are never remembered in a user data structure; and
6. Barring an error, the chain of processing initiated by the call to 3-LISP is never broken (i.e., it will never return).

It is a relatively straightforward final step to translate such a program into one's favourite imperative language.

6 Conclusions

It is widely known that complex issues arise in the implementation of more traditional languages: we have already mentioned a system's treatment of calls between compiled and interpreted code; micro-code routines that call macro-code routines as subroutines are a similar example of implicit level-shifting. The general question of mediating between implementation structures and user structures, and the attendant complexities when they are in different languages, arises in other contexts as well, as for example in SMALLTALK-80's explicit use of a compiled code interpreter for debugging purposes. It is also common experience that providing users with access to implementation structures, although powerful for certain purposes, tends to make an implementation unmodular and difficult to transport onto other architectures.

In (Smith 82a) it was claimed that the reflective capabilities of 3-LISP provide programmers with the power that is normally provided only by giving them access to the underlying implementation. We claimed, in other words, that the full power of implementation access was compatible with a fully abstract, implementation-independent language. In this paper, in showing how to implement such a reflective language, such notions as level-shifting, reifying implicit continuation structures, and so forth, make clear what it is that standard implementations do when they provide those sorts of facilities. In this sense, a level-shifting implementation processor for a procedurally reflective language can be viewed as a *rational reconstruction of implementation more generally*, just as reflection itself can be viewed as a rational reconstruction of the complex programming techniques that use such implementations.

Epilogue and Acknowledgements

Although our first implementation of 3-LISP was based very closely on the techniques described in this paper, we have since shifted to a run-time incremental compiler, that translates 3-LISP code into byte codes for an underlying SECD[x] machine. The resulting system, implemented in InterLISP-D, yields a performance almost exactly the same as that provided by the InterLISP-D interpreter (i.e., 3-LISP programs run about as fast as interpreted InterLISP-D programs). The arguments presented in this paper, coupled with this experience, lead us to believe that although it is tricky, reflection is not an inherently inefficient construct to add to a programming language.

We would like to thank Austin Henderson, Mike Dixon, Dan Friedman, Hector Levesque, and Greg Nuyens for their helpful comments on an early draft. This research was conducted in the Intelligent Systems Laboratory at Xerox Palo Alto Research Center (PARC), as part of the Situated Language Program of Stanford's Center for the Study of Language and Information.

References

[Allen, 1978] John R. Allen, *Anatomy of* LISP, McGraw-Hill, 1978.

[Henderson, 1980] Peter Henderson, *Functional Programming, Application and Implementation*, Prentice-Hall, 1980.

[McCarthy et al., 1965] John McCarthy, et al., LISP 1.5 *Programmer's Manual*, MIT Press, 1965.

[Muchnick & Pleban, 1980] Steven S. Muchnick and Uwe F. Pleban, "A Semantic Comparison of LISP and SCHEME", 1980 LISP Conference, Stanford, 1980.

[Smith, 1982a] Brian C. Smith, "Reflection and Semantics in a Procedural Language", *M.I.T. Laboratory for Computer Science Report MIT-TR-272, 1982.*

[Smith, 1982b] Brian C. Smith, "The Computational Metaphor," available from the author, 1982.

[x] The SECD machine (for "Stack, Environment, Code, Dump"—its internal registers) is an abstract virtual machine, originally designed by Peter Landin to evaluate lambda calculus expressions, which became a standard target for compilers of functional programming languages.

[Smith, 1984] Brian C. Smith, "Reflection and Semantics in LISP," 1984 *ACM POPL Conference,* Salt Lake City, Utah, January 1984 (included here as chapter ■■).

[Smith & des Rivieres, 1984] Brian C. Smith and Jim des Rivières, *Interim 3-LISP Reference Manual,* Xerox Palo Alto Research Center, Intelligent Systems Laboratory Report ISL-l, June 1984.

[Steele, 1976] Guy L. Steele, Jr., "LAMBDA: The Ultimate Declarative", *M.I.T. Artificial Intelligence Laboratory Memo AIM-379, 1976.*

[Steele, 1977a] Guy L. Steele, Jr., "RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization)", *M.I.T. Artificial Intelligence Laboratory Technical Report AI-TR-474, 1977.*

[Steele, 1977b] Guy L. Steele, Jr., "Debunking the "Expensive Procedure Call" Myth", *M.I.T. Artificial Intelligence Laboratory Memo AIM-443, 1977.*

[Steele & Sussman, 1976] Guy L. Steele, Jr. and Gerald 1. Sussman, "LAMBDA: The Ultimate Imperative", *M.I.T Artificial Intelligence Laboratory Memo AIM-*353, 1976.

[Steele & Sussman, 1978a] Guy L. Steele, Jr. and Gerald J. Sussman, "The Revised Report on SCHEME, A Dialect of LISP ", *M.I.T Artificial Intelligence Laboratory Memo AIM-452, 1978.*

[Steele & Sussman, 1978b] Guy L. Steele, Jr. and Gerald 1. Sussman, "The Art of the Interpreter, or, The Modularity Complex (Parts Zero, One, and Two)", *M.I.T Artificial Intelligence Laboratory Memo AIM-453,* 1978.

[Steele & Sussman, 1980] Guy L. Steele, Jr. and Gerald J. Sussman, "Design of a LISP-Based Microprocessor", *Communications of the ACM,* vol. 23, No. 11, November 1980.

[Stoy, 1977] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* MIT Press, 1977.

[Sussman, Holloway, Steele & Bell, 1981] Gerald 1. Sussman, Jack Holloway, Guy L. Steele, Jr., and Alan Bell, "SCHEME-79—LISP on a Chip", *IEEE Computer,* July 1981.

[Sussman & Steele, 1975] Gerald J. Sussman and Guy L. Steele, Jr., "SCHEME: An Interpreter for Extended Lambda Calculus", *M.I.T. Artificial Intelligence Laboratory Memo AIM-*349, 1975.

Appendix: 3·LISP Implementation Processor

This appendix lists the code for all the procedures required in the 3·LISP implementation processor described in section 5. With very minor exceptions, this program is compatible with the dialect of 3-LISP used in the *Interim 3-LISP Reference Manual* [Smith & des Rivieres 84].

```
(define 3-LISP
   (lambda simple []
      (&&read-normalise-print (initial-tower 2) 1 global)))
```

The implementation of READ-NORMALISE-PRINT is similar to the RPP version, except that an explicit procedure implements the REPLY continuation:

```
(define &&READ-NORMALISE-PRINT
   (lambda simple [state level env]
      (&&normalise state (prompt&read level) env
          (make-reply-continuation level env))))

(define &&REPLY-CONTINUATION
   (lambda simple [state result level env]
      (block (prompt&reply result level)
             (&&read-normalise-print state level env) )))
```

The implementation of NORMALISE is virtually identical to NORMALISE itself, except that it must &&CALL continuations, and use implementation version of other PPPs. Similarly, the implementation of REDUCE is similar to REDUCE itself, except that explicit procedures are used to implement both the PROC and ARGS continuations.

```
(define &&NORMALISE
   (lambda simple [state exp env cont]
      (cond [(normal exp) (&&call state cont exp)]
            [(atom exp) (&&call state cont (binding exp env))]
            [(rail exp) (&&normalise-rail state exp env cont)]
            [(pair exp)
             (&&reduce state (car exp) (cdr exp) env cont)])))

(define &&REDUCE
   (lambda simple [state proc args env cont]
      (&&normalise state proc env
          (make-proc-continuation proc args env cont))))
```

```
(define &&PROC-CONTINUATION
   (lambda simple [state proc! proc args env cont]
      (if (reflective proc!)
           (&&call state ↑(de-reflect proc!) args env cont)
           (&&normalise state args env
              (make-args-continuation proc! proc args env cont)))))

(define &&ARGS-CONTINUATION
   (lambda simple [state args! proc! proc args env cont]
      (if (directly-implemented proc!)
          (&&call state cont ↑(↓proc! . ↓args!))[x]
          (&&expand-closure state proc! args! cont))))
```

The implementation of EXPAND-CLOSURE is like the regular EXPAND-CLOSURE code, except we can absorb (shift-down) on PPPs and PPCs—see the discussion in section 5.2. The following checks for NORMALISE and the PPCs:

```
(define &&EXPAND-CLOSURE
   (lambda simple [state proc! args! cont]
      (cond [(and (= (ppp-type proc!) 'normalise)
                  (plausible-arguments-to-normalise args!))
             (&&normalise (shift-down cont state)
                ↑(1st args!) ↑(2nd args!) ↑(3rd args!))]
           [(and (ppc proc!)
                 (plausible-arguments-to-a-continuation args!))
             (&&call-ppc (shift-down cont state)
                        ↑proc!
                        ↑(1st args!))]
           [$t (&&normalise state
                         (body proc!)
                         (bind (pattern proc!)
                               args!
                               (environment proc!))
                         cont)]))))
```

The implementation of NORMALISE-RAIL is similar to NORMALISE-RAIL itself, except that explicit procedures are used to implement both the FIRST and REST continuations.

```
(define &&NORMALISE-RAIL
   (lambda simple [state rail env cont]
      (if (empty rail)
          (&&call state cont (rcons))
          (&&normalise state (1st rail) env
              (make-first-continuation rail env cont)))))
```

[x] In the published paper this line was erroneously printed as
    (&&call state cont ↑(↑proc! . ↑args!))

```
(define &&FIRST-CONTINUATION
   (lambda simple [state first! rail env cont]
      (&&normalise-rail state (rest rail) env
          (make-rest-continuation first! rail env cont)))

(define &&REST-CONTINUATION
   (lambda simple [state rest! first! rail env cont]
      (&&call state cont (prep first! rest!)))))
```

LAMBDA and IF must be implemented as primary processor procedures, IF with an explicit procedure in place of its normal continuation:

```
(define &&LAMBDA
   (lambda simple [state [kind pattern body] env cont]
      (&&call state cont (ccons kind ↑env pattern body)))

(define &&IF
   (lambda simple [state [premise c1 c2] env cont]
      (&&normalise state premise env
          (make-if-continuation premise c1 c2 env cont))))

(define &&IF-CONTINUATION
   (lambda simple [state premise! premise c1 c2 env cont]
      (&&normalise state (ef ↓premise! c1 c2) env cont))))
```

(&&CALL f a1 ... ak) would be like (f a1 ... ak) except that if f is a PPP or PPC, the corresponding implementation version is used instead; if f is directly implemented, we use the implementation directly and then shift up; otherwise we shift up and do an explicit expand closure one level higher.

```
(define &&CALL
   (lambda simple x
      (let [[state (1st x)] [f (2nd x)] [a (rest (rest x))]]
         (cond [(ppp ↑f) (&&call-ppp state f a)]
               [(ppc ↑f) (&&call-ppc state f (1st a)]
               [(directly-implemented ↑f)
                (&&call (shift-up state)
                        (reify-continuation state)
                        ↑(f . a))]
               [$t (&&expand-closure (shift-up state)
                       ↑f ↑a (reify-continuation state))]))))

(define &&CALL-PPP
   (lambda simple [state f a]
      ((select (ppp-type ↑f)
          ['normalise &&normalise]
          ['normalise-rail &&normalise-rail]
          ['reduce &&reduce]
          ['read-normalise-print &&read-normalise-print]
          ['if &&if]
```

```
        ['lambda &&lambda])
      . (prep state a))))

(define &&CALL-PPC
   (lambda simple [state f arg]
      (select (ppc-type ↑f)
         ['proc (&&proc-continuation state arg (ex 'proc f)
            (ex 'args f) (ex 'env f) (ex 'cont f))]
         ['args (&&args-continuation state arg (ex 'proc! f)
            (ex 'proc f) (ex 'args f) (ex 'env f) (ex 'cont f)))]
         ['first (&&first-continuation state arg (ex 'rail f)
            (ex 'env f) (ex 'cont f))]
         ['rest (&&rest-continuation state arg (ex 'first! f)
            (ex 'rail f) (ex 'env f) (ex 'cont f))]
         ['reply (&&reply-continuation state arg (ex 'level f)
            (ex 'env f))]
         ['if (&&if-continuation state arg (ex 'premise f)
            (ex 'c! f) (ex 'c2 f) (ex 'env f) (ex 'cont f))]])))
```

The next six MAKE-XXX-CONTINUATION procedures look very messy, but they are really trivial: all they do is to construct a closure that is identical to the type of closure that would have been constructed by the RPP, had it been running instead of this implementation. These continuations are only used to fake the RPP; their only use here is as templates for later recognition. EX(TRACT) is used to extract bindings for variables that were enclosed in these faked continuations.

```
(define MAKE-PROC-CONTINUATION
   (lambda simple [proc args env cont]
      ↑(ccons 'simple ↑(bind '[proc args env cont reduce]
                             ↑[proc args env cont reduce]
                             global)
         '[proc!]
         '(if (reflective proc!)
              (↓(de-reflect proc!) args env cont)[x]
              (normalise args env
                 (lambda [args!]
                    (if (primitive proc!)
                        (cont ↑(↓proc! . ↓args!))
                        (normalise (body proc!)
                                   (bind (pattern proc!)
                                         args!
                                         (environment proc!))
                                   cont))))))))
```

[x] The '↓' in this line was '↑' in the published paper, but that was an error.

```
(define MAKE-ARGS-CONTINUATION
   (lambda simple [proc! proc args env cont]
      ↓(ccons 'simple
               ↑(bind '[proc! proc args env cont reduce]
                       ↑[proc! proc args env cont reduce]
                       global)
               '[args!]
               (if (primitive proc!)
                   (cont ↑(↓proc! . ↓args!))
                   (normalise (body proc!)
                              (bind (pattern proc!)
                                    args!
                                    (environment proc!))
                              cont)))))

(define MAKE-FIRST-CONTINUATION
   (lambda simple [rail env cont]
      ↓(ccons 'simple
               ↑(bind '[rail env cont normalise-rail]
                       ↑[rail env cont normalise-rail]
                       global)
               '[first!]
               '(normalise-rail (rest rail) env
                   (lambda [rest!]
                      (cont (prep first! rest!)))))))

(define MAKE-REST-CONTINUATION
   (lambda simple [first! rail env cont]
      ↓(ccons 'simple
               ↑(bind '[first! rail env cont normalise-rail]
                       ↑[first! rail env cont normalise-rail]
                       global)
               '[rest!]
               '(cont (prep first! rest!)))))

(define MAKE-REPLY-CONTINUATION
   (lambda simple [level env]
      ↓(ccons 'simple
               ↑(bind '[level env read-normalise-print]
                       ↑[level env read-normalise-print]
                       global)
               '[result]
               '(block (prompt&reply result level)
                       (read-normalise-print level env)))))

(define MAKE-IF-CONTINUATION
   (lambda simple [premise c1 c2 env cont]
      ↓(ccons 'simple
               ↑(bind '[premise c1 c2 env cont if]
                       ↑[premise c1 c2 env cont if]
                       global)
               '[premise!]
               '(normalise (ef ↓premise! c1 c2) env cont))))
```

```
(define EX
   (lambda simple [variable function]
      ↓(binding variable (environment ↑function))))
```

Various utilities dealing with state management and continuations for each level:

```
(define INITIAL-TOWER
   (lambda simple [level] (scons level))

(define SHIFT-DOWN
   (lambda simple [continuation state]
      (prep continuation state)))

(define REIFY-CONTINUATION
   (lambda simple [state]
      (if (= (length state) 1)
          (make-reply-continuation (1st state) global)
          (1st state))))

(define SHIFT-UP
   (lambda simple [state]
      (if (= (length state) 1)
          (scons (1+ (1st state)))
          (rest state))))
```

Predicates to check the plausibility of arguments, closures, and environments, to be used preparatory to shifting down and using implementation versions:

```
(define PLAUSIBLE-ARGUMENTS-TO-A-CONTINUATION
   (lambda simple [args!]
      (and (rail args!)
           (= (length args!) 1)
           (handle (1st args!)))))

(define PLAUSIBLE-ARGUMENTS-TO-NORMALISE
   (lambda simple [args!]
      (and (rail args!)
           (= (length args!) 3)
           (handle (1st args!))
           (plausible-environment-designator (2nd args!))
           (plausible-continuation-designator (3rd args!)))))

(define PLAUSIBLE-ENVIRONMENT-DESIGNATOR
   (lambda simple [env!]
      (and (rail env!)
           (or (= env! ↑global)
               (empty env!)
               (and (plausible-binding-designator (1st env!))
                    (plausible-environment-designator
                       (rest env!)))))))
```

```
(define PLAUSIBLE-BINDING-DESIGNATOR
   (lambda simple [b!]
      (and (rail b!)
            (= (length b!) 2)
            (handle (1st b!)
            (atom ↓(1st b!)
            (handle (2nd b!))))

(define PLAUSIBLE-CONTINUATION-DESIGNATOR
   (lambda simple [c!]
      (and (closure c!)
           (not (reflective c!))
           (or (atom (pattern c!))
               (and (rail (pattern c!))
               (= 1 (length (pattern c!))))))))
```

Predicates defined over closures, sorting them into the various types that the implementation needs to know about: PPPs, PPCs, etc. Also, there are utilities for recognizing closures of these various types.

```
(define DIRECTLY-IMPLEMENTED
   (lambda [closure]
      (or (primitive closure)
          (kernel-utility closure))))

(define PPP
   (lambda simple [closure]
      (not (= 'unknown (ppp-type closure)))))

(define PPP-TYPE
   (lambda simple [closure]
      (identify-closure closure *ppp-table*)))

(set *PPP-TABLE*
   [['normalise ↑normalise]
    ['reduce ↑reduce]
    ['normalise-rail ↑normalise-rail]
    ['read-normalise-print ↑read-normalise-print]
    ['lambda (de-reflect ↑lambda)]
    ['if (de-reflect ↑if)]])

(define PPC
   (lambda simple [closure]
      (not (= 'unknown (ppc-type closure)))))

(define PPC-TYPE
   (lambda simple [closure]
      (identify-closure closure *ppc-table*)))
```

```
(set *PPC-TABLE*
   [['proc  ↑(make-proc-continuation  '? '?  '?  '? )]
    ['args  ↑(make-args-continuation  '?  '?  '?  '?  '? )]
    ['first ↑(make-first-continuation '?  '?  '? )]
    ['rest  ↑(make-rest-continuation  '?  '?  '?  '? ) ]
    ['reply ↑(make-reply-continuation '?  '? ) ]
    ['if    ↑(make-if-continuation    '?  '?  '?  '?  '? )]]])

(define KERNEL-UTILITY
   (lambda simple [closure]
      (member closure *kernel-utility-table*)))

(set *KERNEL-UTILITY-TABLE*
   [↑1st         ↑double       ↑normal       ↑rail
    ↑2nd         ↑environment  ↑normal-rail  ↑rebind
    ↑atom        ↑external     ↑pair         ↑reflective
    ↑bind        ↑handle       ↑primitive    ↑rest
    ↑binding     ↑length       ↑prompt&read  ↑unit
    ↑de-reflect  ↑member       ↑prompt&reply ↑vector-constructor])

(define IDENTIFY-CLOSURE
   (lambda simple [closure table]
      (cond [(empty table) 'unknown]
            [(similar-closure closure (2nd (1st table)))
             (1st (1st table))]
            [$T (identify-closure closure (rest table))] I))~

(define SIMILAR-CLOSURE
   (lambda simple [closure template]
      (or (= closure template)
          (and (isomorphic (pattern closure) (pattern template))
               (isomorphic (body closure) (body template))
               (= (reflective closure) (reflective template))
               (similar-environment (environment closure)
                                    (environment template))))))

(define SIMILAR-ENVIRONMENT
   (lambda simple [environment template]
      (or (= ↑environment ↑template)
          (and (empty environment) (empty template))
          (and (not (empty template))
               (not (empty environment))
               (= (1st (1st environment)) (1st (1st template)))
               (or (= ''? (2nd (1st template)))
                   (= (2nd (1st environment))
                      (2nd (1st template))))
               (similar-environment (rest environment)
                                    (rest template))))))
```

— Were this page been blank, that would have been unintentional —

5 — Varieties of Self-Reference[†]

Abstract

The significance of any system of explicit representation depends not only on the immediate properties of its representational structures, but also on two aspects of the attendant circumstances: implicit relations among, and processes defined over, those individual representations, and larger circumstances in the world in which the whole representational system is embedded. This relativity of representation to circumstance facilitates local inference, and enables representation to connect with action, but it also limits expressive power, blocks generalisation, and inhibits communication. Thus there seems to be an inherent tension between the effectiveness of located action and the detachment of general-purpose reasoning.

It is argued that various mechanisms of *causally-connected self-reference* enable a system to transcend the apparent tension, and partially escape the confines of circumstantial relativity. As well as examining self-reference in general, the paper shows how a variety of particular self-referential mechanisms—autonymy, introspection, and reflection—provide the means to overcome specific kinds of implicit relativity. These mechanisms are based on distinct notions of self: self as unity, self as complex system, self as independent agent. Their power derives from their ability to render explicit what would otherwise be implicit, and implicit what would other-

[†]Original published in Joseph Y. Halpern (ed.), *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference*, Monterey, California, March 19–22. Los Altos, California: Morgan Kaufmann: 1986, pp. 19–43. Also available as CSLI Technical Report CSLI–87–76, Center for the Study of Language and Information, Stanford, California, 1986.

wise be explicit, all the while maintaining causal connection between the two. Without this causal connection, a system would either be inexorably parochial, or else remain entirely disconnected from its subject matter. When appropriately connected, however, a self-referential system can move plastically back and forth between local effectiveness and detached generality.

1 Introduction

"If I had more time, I would write you more briefly." So, according to legend, said Cicero—thereby making reference to himself in three different ways at once. First, he quite explicitly referred to himself, in the sense of naming himself (with the word 'I') as part of his subject matter. Second, his sentence has content, or conveys information, only when understood "with reference to him"—specifically, with reference to the circumstances of his utterance. To see this, note that if I were to use the same sentence right now I would say something quite different (something, for example, that might lead you to wonder whether this paper might not have been shorter). Similarly, the pronoun 'you' picks someone out only relative to Cicero's speech act; the present tense aspect of 'had' gets at a time two millennia ago; and so on and so forth. Third, as well as referring to himself in these elementary ways, he also said something that reflected a certain *understanding* of himself and of his writing, enabling him to make a claim about how he would have behaved, had his circumstances differed.

In spite of all these self-directed properties, though, there is something universal about Cicero's statement as well, transcending what was particular to his situation. It is exactly this universality that has led the statement to survive. So we might say in summary that Cicero *referred to himself,* that the content of his statement was *self relative,* that he expressed or manifested *self understanding,* and yet that, in spite of all of these things, he managed to say something that did not, ultimately, have much to do with himself at all.

Or we might like to say such things, if only we knew what those phrases meant. One problem is that thay all talk about the familiar, but not very well-understood, notion of 'self'. Perry (1983) has claimed that the self is so "burdened by the history of philosophy" as to almost have been abandoned by that tradition (though his own work, on which I will depend in the first two sections, is a notable exception). Researchers in Artificial Intelligence (AI), however, have rushed in with characteristic fearlessness and tackled self-reference head-on. AI's interest in the self is not new: dreams of self-understanding systems have permeated the field since its earliest days. Only recently, however, has this general interest given way to specific analyses and proposals. Technical reports have begun to appear in what we can informally divide into three traditions. The first., which (following Moore) I will call the **autoepistemic** tradition, has emerged as part of a more general investigation into reasoning about knowledge and belief (the theme of this conference). A second more procedural tradition, focusing on so-called meta-level reasoning and inference about control, is illustrated by such systems as FOL[1] and 3-l.isp:[2] for discussion I will call this the **control** camp. Finally, in collaboration with the philosophical and linguistic communities, what I will call *the* **circumstantial** tradition in AI has increasing come to recognise the pervasiveness of the self-relativity of thought and language (self-reference in the sense of "with reference to self").[3]

In spite of all this burgeoning activity, two problems have not been adequately addressed.

The first problem is obvious, though difficult: while many particular mechanisms have been proposed, no clear, single concept of the self has emerged, capable of unifying all the disparate efforts. Technical results in the three traditions overlap surprisingly little, for example, in spite of their apparently common concern. Nor has

[1] «Ref»

[2] «Ref»

[3] For examples of the autoepistemic tradition, see for example Fagin & Halpern (1985), Konolige (1985), Levesque (1984), Moore (1983), and Perlis (1985). For the control tradition, see Batali (1983), Bowen & Kowalski (1982), Davis (1976), Davis (1980), de Kleer et al. (1979), des Rivières and Smith (1984), Doyle (1980), Friedman and Wand (1984), Genesereth and Smith (1982), Hayes (1973), Laird and Newell (1983), Laird et al. (forthcoming), Smith (1982), Smith (1984), and Weyhrauch (1980). For the circumstantial tradition, see Kaplan (1979), Barwise and Perry (1983), Perry (1985a), Perry (1985b), Perry (forthcoming), and Rosenschein (1985). Finally, I should mention those who have studied self-reference in specific cognitive tasks: for example Collins (1975) and Lenat and Brown (1984).

the general enterprise been properly located in the wider intellectual context. For example, as well as exploring the *self* we should understand what sort of *reference* self-reference involves, and how it relates to reference more generally. Also, it has not been made clear how the inquiries just cited relate to the self-referential puzzles and paradoxes of logic (which, for discussion, I will call **narrow self-reference**). At first glance the two seem rather different: AI is apparently concerned with reference to *agents*, not to sentences, for starters—and with whole, complex selves, not individual utterances or even beliefs. We are interested in something like the lay, intuitive notion of "self" that we use in explaining someone's actions by saying that they lack self-knowledge. It is not obvious that there is anything even circular, let alone paradoxical, about this familiar notion (folk psychology does not go into any infinite loops over it). And yet we will uncover important similarities having to do with limits.

The second problem is more pointed: there seems to be a contradiction lurking behind all this interest in self-reference. The real goal of AI, after all, is to design or understand systems that can reason about the *world*, not about *themselves*. Who cares, really, about a computer's sitting in the corner referring to itself? Like people, computers are presumably useful to the extent that they participate with us in our common environment: help us with finances, control medical systems, etc. Introspection, reflection, and self-reference may be intriguing and incestuous puzzles, but AI is [fundamentally] a pragmatic enterprise. Somehow—in ways that no one has yet adequately explained—self-reference must have some connection with full participation in the world.

In this paper I will attempt to address both problems at once, claiming that the deep regularities underlying self-reference arise from necessary architectural aspects of any embedded system. Both cited problems arise from our failure to understand this—a failure attributable in part to our reliance on restricted semantical techniques, particularly techniques borrowed from traditional mathematical logic, that ignore circumstantial relativity. Once we can see what problem the self is "designed to solve", we will be able to integrate the separate traditions, and explain the apparent contradiction.

The analysis will proceed in three parts. First, in section 2 I will assemble a framework in terms of which to understand both self and self-reference, motivated in part by the technical proposals just cited. The major insights of the circumstantial tradition will be particularly relevant here. Second, in section 3, I will sketch a tentative analysis of the structure of the circumstantial relativity of any representational system. This specificity will be necessary in order to ground the third, more particular analysis, presented in section 4, of a spectrum of self-referential mechanisms. Starting with the simple indexical pronoun 'I', and with unique identifiers, I will examine assumptions underlying the autoepistemic tradition, moving finally to canvass various models of introspection and reflection that have developed within the control camp.

The way l will resolve the contradiction is actually quite simple. It is suggested by my inclusion of *self-relativity* right alongside genuine *self-reference.* Some readers (semanticists, especially) may suspect that this is a pun, or even a use/mention mistake. But in fact almost exactly the opposite is true. [It is a fundamental thesis underlying the present analysis that] the two notions are intimately related, forming something of a complementary pair. Time and again we will see how an increase in the latter (self-reference) enables a decrease in the former (self-relativity). For fundamental reasons of efficiency, all organisms must at the ground level be tremendously self-relative.[a] On the other hand, although it enables action, this [basic] self-relativity inhibits cognitive expressiveness, proscribes communication, restricts awareness of higher level generalisations, and generally interferes with the agent's attaining a variety of otherwise desirable states. *The role of self-reference, [it will be argued,] is to compensate for this parochial self-relativity, while retaining the ability to act,*

Explicit self-reference, that is, can provide an escape from implicit self-relativity.

Intuitively, it is easy to see why. Suppose, upon hearing a twig break in the woods, I shout "There is a bear on the right!" My meaning would be perfectly clear, but I have explicitly mentioned only one of the four arguments involved in the TO-THE-RIGHT-OF

[a] «Talk about this as a precursor to the deixis adumbrated in O3»

relation;[4] *the* other three remain implicit and self-relative, determined by circumstance. However I can lessen the degree of implicit self-relativity by mentioning some of the other arguments explicitly. Look at this as a two stage process: one to get rid of the implicitness, one to get rid of the self-relativity (implicitness and self-relativity, that is, are distinct; *both* characterise ground-level action). In particular, the first move is to shift from the original statement to another that has roughly the same content, but that makes another argument explicit: "There is someone to the right *of me*." This latter statement is still self-relative, of course, but in a different, explicit, way. Now that I have a place for another argument, I can make the second move, and use a different expression to refer to someone else: "There is someone to the right *of you,*" or "There is someone to the right *of us all*."

Thus the self provides a **fulcrum**, allowing a system to shift in and out of the particularities of its local situation. Both directions of mediation are necessary: neither totally local relativity, nor completely detached generality, would be adequate on its own. Roughly, the first would enable you to act, but thoughtlessly; the second, to think, but ineffectively.

So there is really no contradiction, after all. But there is some irony: the self is the source of the problem, as well as being an ingredient in the solution. The overall goal in attaining detached general-purpose reasoning is to *flush the self from the wings*. However, the way to do that is first to drag it onto center stage. If you were to stop there, then you really would be stuck with a contradiction—or at least with a system so self-involved it could not reason about the world at all. Fortunately, however, once the self is brought into explicit view, it can then be summarily dismissed.

2 Circumstance, Self, and Causal Connection

«Put in an introductory sentence or three … »

…

[4] The fourth is orientation. Even if you and I are in essentially the same place, and looking out in the same direction, and if *A* is to the right of *B* from my point of view, *A* will nonetheless be to the left of *B* from your point of view. if you happen to be standing on your head. Gravity establishes such a universal orientation that we rarely need to make this [final?] circumstantially determined argument position explicit.

2a Assumptions

I will focus on *representational* systems—without defining them, though I will assume they include both people and computers, at least with respect to what we would intuitively call their linguistic, logical, or rational properties. For a variety of reasons I will not insist that representational systems be ‘syntactic’ or ‘formal’ (although what I have to say would equally well apply under what people take to be that conception).[5] Several other assumptions, however, will be important.

First, I take it that systems do not represent as indivisible wholes, in single representational acts, but in some sense have representational *parts*, each of which can be said to have content at least somewhat independently (what content a part has, however, will often depend on all the other parts—i.e., the parts do not need to be *semantically* independent). I take this notion of ”part” very broadly: parts might be internal structures (tokens of mentalese, data structures, whatever), distinct utterances or discourse fragments issued over time, or even different aspects or dimensions of a complex mental state (what Perry has informally called mental “counties”). I will use ‘agent’ or ‘system’ to refer to a representational system as a whole, and ‘representational structure’ to refer to [such] ingredients. When I specifically want to focus on the internal structures that are causally responsible for an agent’s or system’s actions, however, I will talk of **impressions** (as opposed to **expressions**, which I take to be tokens or utterances, *external* to an agent, in a consensual [or communicative] language). Impressions are meant to include data structures, elements of a knowledge representation system, or aspects of a total mental state. Such structures are sometimes classified abstractly (particularly in [computer science’s] “abstract data type” tradition), or identified with other abstract things to which they are thought to be isomorphic (like *beliefs*), but I will refer to them directly, because of my architectural bias and interest in causal role.

Second, [as well as severally constituting a complex system or agent as a whole,] representational structures are themselves likely

[5] [I set formality aside] primarily because, [in spite of prevailing consensus,] I do not think the notion is in fact coherently applicable to computation. See [Smith forthcoming (a)].

to be compositionally constituted, which just means that they too may have parts (nothing is being said about compositional semantics—at least not yet). Again, the notion of part is rough: imagine something like a grammatical structure, or set of partially independent properties or elements, each of which contributes to the meaning of the whole. Utterances constituted of words according to the dictates of grammar are one example; composite structures in a data or "knowledge" base are another. Thus the words 'I', 'would', 'have', and so on, are components of Cicero's claim (at least in its English translation). Since the term 'element' is biased towards ingredient objects and away from features or characteristics, and 'property' is biased the other way, I will refer to such parts as **aspects** of a structure or impression.

Finally, each constituent will be assumed to have what philosophers would call a *meaning* which is something, probably abstract, that indicates just what and how it contributes to [what I will call] the *interpretive content* of the composite wholes in which it participates (i.e., I mean now to embrace just about the weakest form of compositional semantics I can imagine). Meaning [in this sense] is not, typically, the same as [interpretive] content; rather, it is something that plays a role in giving a representation, or a use of a representation, whatever [interpretive] content it has. So the meaning of the word 'Caitlyn' might be something like a relation between speakers and the world, a relation that enables those speakers, when they use the word, thereby to refer to whomever has that particular name in the overall situation being described. Though it is ultimately untenable, one can think of meaning as something a representational structure has "on its own", so to speak, in the sense of being independent of context of use; the [interpretive] content arises only when it is used, in a full set of circumstances. So 'I' means the same thing when different people use it, but those uses have different [interpretive] contents—[you when you use it, I when I do].

As well as distinguishing meaning and content, we need to distinguish the latter—roughly, what a representation or statement is about—from an even wider notion of [general] semantical **significance**, where the latter is taken to include not only the content but the full conceptual or functional role that the representational structure can play in and for the agent.[6] So for example in a computer implementation of a natural deduction system for traditional logic, a formula's content might be taken to be its standard (model-theoretic) interpretation, whereas its full significance would include its proof-theoretic role as well. It is distinctive of standard logical systems to view a sentence's meaning as the sole determiner of its content, and to take content as independent of any other aspect of significance. Situation theory[7] distinguishes *meaning* and *content*, and admits the dependence of the latter on circumstance, but takes both as specifiable independent of conceptual or functional role. In some of the cases we will look at, however, such as the use of inheritance mechanisms to implement default reasoning, all three will be inextricably intertwined.[c]

2b Circumstantial Relativity

The first and most important observation we can make about representational systems in terms of these distinctions[d] is that a great deal of the full significance of a representational system will not, in general, be directly or explicitly represented by any of the representational structures of which it is composed. Instead, [that additional significance] will be contributed by the attendant circumstances. Section 3 will be devoted to saying what "attendant circumstances" might mean, but some familiar examples will illustrate the basic intuition. As we have already seen, whom the word 'I' refers to is not indicated on the word itself, nor is it part of the word's meaning; rather, the meaning of 'I', [given the notion of meaning we are using,] is merely that it refers *to whomever says it*—[with the narrowing of that generic meaning to a particular individual settled by the particularities of the saying.] Similarly, the referent of a pronoun may be determined by the structure and circumstances of the conversation in which it is used. If I say "So-

[6] The term "conceptual role" is associated with Harman; see Harman (1982), and Smith (1984) for a computational account treating both content and conceptual role simultaneously.

[7] See Barwise & Perry (1983).

[c] «That paragraph is extraordinarily dense; admit this, and maybe say something about what it means? A sidebar?»

[d] «I.e., the ability to refer to the compositional contribution to meaning, content, and full significance of mereological impressions.»

lar tax credits have been extended for a year," the year in question, and the temporal constraints I place on it by using the past tense, emerge from the time of my utterance, not from anything explicit in the [meaning of the] words. And, to take perhaps the ultimate example, whether what I say is *true*—which is, after all, part of its significance—is determined by the world, not (at least typically) by anything about the sentence itself.

Similarly, as the Carroll paradoxes show,[e] the fundamental rules of inference cannot themselves emerge in virtue of being explicitly represented, because further or deeper rules of inference would be required in order to use them. Nor do even the so-called "eternal" sentences of mathematics and logic carry all of their significance on their sleeve. [While relevant to their semantical contribution, the syntactic category of lexical items in logical formulae is not explicitly represented:] that a predicate letter is a predicate letter is true in, but is not represented by, that formula. Similarly, Lisp's being dynamically scoped is not explicitly represented in Lisp; [the same holds for the order of argument evaluation—left-to-right or right-to-left[e]]. Or take the inheritance example suggested above: suppose you implement a representation system where a (representation of a) property attached to a node in a taxonomic lattice is taken to mean "an object of this type should be taken to have this property unless there is more specific evidence to the contrary." Thus, to use the standard example, if an impression of FLIES(*x*) is attached to the BIRD node, then the system is wired to "believe" that a particular bird will fly so long as there is not an impression of ¬FLIES(*x*) attached in the lattice between the BIRD node and the individual node representing the bird in question. In such a system the content (not meaning!) of the "so long as there is not…" part of the impression's meaning is architecturally determined: it is an implicit part of the overall system's structure, not explicitly represented, and it depends on the surrounding circumstances that obtain throughout the rest of the system, not on anything local to the particular structure under consideration.

This last example is intended to suggest why I am not distin-

[e] «They should probably be explained: maybe with reference to the *Alice in Wonderland* case?»

[e‘] «Maybe note that it is not even revealed by standard meta-circular interpreter code.»

guishing internal circumstance (whether there are other impressions standing in certain relational properties with a given one, say) and external circumstance (who is talking, where the agent is located, etc.). An informal division between the two will be introduced in section 3, but the similarities are more important than the differences, as evidenced in the similarities of mechanisms to cope with them. For one thing, since activity has to arise, ultimately, from the local interaction of parts, it may not matter whether a part's relational partner is somewhere across the system, or outside in the world; what will matter is that it is not right "here." Perhaps more significantly, the internal/external distinction is far from clean: since agents are part of the world in which they are embedded, some properties cross the boundary. For example, the passage of so-called "real time" is often as crucial for internal mechanism as for overall agent.

2c Efficiency

Before trying to carve circumstantial relativity into some coherent substructure, it helps to understand why it is so pervasive. The answer has to do with efficiency, in a broad sense of that term. Specifically, in order for a finite agent to survive in an indefinitely variable world, it is important that multiple uses of its parts or aspects have different consequences, each appropriate to how the world is at that particular moment. Partly this enables a system to avoid drowning in details: any facts that are persistent across its experience can be "designed out," so to speak, and carried by the environment (as gravity carries the orientation argument for the human notion of to-the-right-of). But efficiency goes deeper, having also to do with how to cope with genuinely different situations.

The point is easiest to see in the case of action, where it is in fact so obvious as to be almost banal. Specifically, different occurrences of what we take to be the "same" action have different consequences, depending on the circumstances of the world in which they take place. So if I take a scoop with my backhoe, what I pick up in its shovel will depend not on my action as such, but on the ground behind my tractor. Thus l can perfectly coherently say things like "after doing the same thing over and over, l suddenly cut the telephone cable." I.e., one can imagine viewing an action (read: *mean-*

ing) as a relation between a local flexing of the tractor's appendages and the situation in which that flexing takes place. The consequences of the action in a given situation (read: *content*) can be determined by applying the relation to the situation itself.

Our conception of actions works in this way because any other way of "parsing" it would be devastatingly inefficient. Each day we want our actions to lead to different consequences (eating new meals, for example); it would be a terrible strain if we had to be structured differently for each one (to say nothing of: a terrible strain if we had to describe the way we were structured each day, in a manner that had to take explicitly into account the metaphysical way in which the new day was different). As it is, we can have (or use) a finite and relatively stable structure, which can locally repeat doing the "same" things; the circumstantial relativity of perception and action will take care of providing the new consequences. The result is an efficient solution to what Perry characterises as a fundamental design problem:[e]

> "Imagine you want to populate the world with animals that will act effectively to meet their needs.
>
> There is one fundamental problem. Since these organisms will be scattered about in different locations, what they should do to meet their needs will depend on where they are and what things are like *around them*. This seems to present a problem. You can't just make them all the same, for you don't want them to do the same thing. You want those in front of nuts to lunge and gobble, and those who aren't to wander around until they are. (I have Grice's squarrels in mind.)
>
> You decide to make them each different…But then it strikes you that there is a more efficient way to do it. You can make them all the same, as long as you are a bit more abstract about it. You can make them all the same, [in the sense of

[e] «Note that this entire discussion—including Perry's—only gestures, rather crudely, at the point, because it makes free use of such constructions as "same" and "different," both of which are defined with respect to *types*, which already build in many of the points being made. To makes these points without some such presupposition is impossible; to say it as carefully as possible, while nevertheless acknowledging that ultimate limitation, would take several pages of exceedingly complex metaphysics.»

> having] their action controlling states depend on where they are. And you can do that, by giving them perception, as long as it is perception of the things about them. That is, you can make their internal states work in terms of what we have called *subject relative conditions and abilities*. You make them each go into state G when they are hungry and there are nuts in front of them, and each lunge forward and gobble when they are in state G.
>
> This way of solving a design problem, we call efficiency."[8]

Like eating, representation needs to be efficient, and for similar reasons. First, actions are required in order to use and profit from the internal impressions: what page a least-recently-used virtual memory system discards, for example, will depend on circumstances. Second, impressions can themselves be circumstantially relative (what Perry calls "subject-relative") as both the pronoun and inheritance examples show. Finally, you would expect *ground-level* representations—representations connected directly with action and perception—to have the same (efficient) relativity as the actions and perceptions with which they are connected. Only in this way is there any hope of giving the connection between representation and action the requisite integrity. It is plausible to imagine a signal on the optic nerve directly engendering a rough impression of THERE-IS-SOMETHING-TO-THE-RIGHT, but implausible to imagine its producing (and even this, of course, is still earth-relative):[f]

RIGHT(SOMETHING, 38°N/120°W, 187°N, GRAVITY-NORMAL,
3-JAN-86/12:40:04)

Similarly, the stomach must first create the grounded, impression "HUNGRY!"; it takes inference to turn this into "Won't you have some more pie?"

2d The Role of the Self

Circumstantial relativity is not something an agent should expect

[8] Perry (1983); second emphasis added.

[f] The arguments of location (38°N/120°W), orientation (187°N), vertical-orientation (GRAVITY-NORMAL), and time (3-JAN-86/12:40:04) held of the author at the moment this paper was written.

to get over, but it [nevertheless] has a down side. First, it does not lend itself to communication, if the relevant circumstances of the two communicators differ. If some agent *A* were simply to give another agent *B* a copy of one of its representational impressions, and *B* were to incorporate it bodily, the result might have completely different significance (and possibly even meaning) from the original. Information would not have been conveyed.[g] If you are facing me, hear me say "There is a bear on the right!", take the sentence as your own, and then leap to *your* left, you would land in trouble.

Second, one of representation's great virtues is that it can empower a system with respect to situations remote in space or time, outside the system's own local circumstances.[g] However, in order to represent those situations using impressions connected to those it uses to control action, the system must at least represent its own relativity, in order to be able to mediate between those less self-relative generalisations and more familiar implicit ones. I.e., to the extent that the content of its representational structures arise from implicit factors, it is impossible for a system to modify, discriminate with respect to, or make different use of any of the implicitly represented aspects of those representations' contents. If "HUNGRY!", without any argument, is the system's only means of representing the property of hunger, then it will not be able to represent any generalisation involving anyone else (such as that the bear on the right is hungry), or anything generic, such as that hunger sharpens the mind.

The third limit arising from circumstantial relativity depends on another fundamental fact about representation: its ability to represent situations in ways other than how they are. I will call this property of representation its partial **disconnection** (thus tree rings, under normal conditions of rainfall, do not quite qualify as representations, on this account, because they are so nomically locked in to what they purportedly represent that they cannot be wrong). A particular case of internal disconnection illustrates the third limit of circumstantial relativity.

Typically, as long as some aspect of its internal architecture is

[g] On the assumption that by 'information' we mean *information content*.

[g] «Put in a pointer to the non-effective relations to the distal that occupy so much of my attention later on.»

not represented, a system will behave in the "standard" way with respect to that aspect. So to consider the inheritance example again, the default FLIES(x) will always be interpreted by the underlying architecture in the "so long as there is not…" way. Suppose, however, that you want a variant on this behaviour: say, that the default should be over-ridden not if any specific information to the contrary is represented, but only if that more specific contrary information has been obtained from a reliable external source. Being implicit, however, the default way of doing things is not available for this kind of modification. But if the internal dependence had been explicitly represented, then (as a consequence of the generative power of representation generally) the appropriate modification of the default behaviour could likely be represented as well. [And then—assuming that *representation* of internal behaviour is causally linked with how and what internal behaviour actual comes about, the modification could take effect.[g]] In this way (under some constraints we will get to in a moment) a system could alter its behaviour appropriately.

In sum, explicit representation of circumstantial relativity paves the way for more flexible behaviour; without it, a system is locked into its primitive ways of doing things.

Among other things, the representation of circumstantial relativity requires the representation of one's self, because that self, [in both its generality and particularity, is almost invariably] the ultimate source of the relativity. There are of course different aspects of self, corresponding to different aspects of relativity: the self as a unity (useful in such cases as TO-THE-RIGHT-OF), the self as a complex organization (applicable to the inheritance example), the self as an agent (relevant to generalising about the consequences of hunger).

Note that merely giving a system an impression that refers to itself does not automatically solve the problem of circumstantial relativity. To see this, imagine installing within a system, as if by surgery, some impressions less self-relative than usual. For example, one might imagine giving a system: (i) a three-place represen-

[g] «Say something about how this "causal connection" becomes a big issue later on—and cite §.»

tation of "to-the-right-of"—say, RIGHT3(*x,y,z*); and (ii) a distinguished token—say, $ME—to use as its own name. Chances are that the provision of such representations would be conceptually possible, in the sense of not being architecturally precluded. They might enable the system or agent to reason (rather like a theorem-proving system) about some world. The problem would be that, without additional machinery, there would be no way for that system to act in that world, were it to find itself suddenly located there—i.e., no way for it to connect an occasioning of RIGHT3 with an occasioning of the grounded THERE'S-8OMETHING-TO-THE-RIGHT!). The experience for the system might be a little like that of students who learn mathematics in a totally formal way (in the derogative sense), being able to manipulate formulae of various shapes around in prescribed ways, with no real sense of what they mean. Merely providing such explicitised representations, and tying them into the system's general reasoning abilities, does not in and of itself make such representations *matter* to the system; they would not thereby be connected with the agent's life [in the way in which the presumed interpretation would imply]. Furthermore, in a more realistic case where surgery is precluded (say, ours), there is no way to see how such representations could *arise* [either phylogenetically or ontogenetically], given that they would have no direct tie to action or perception.

There is a problem, in other words: systems and agents must *connect* any explicit representations of their circumstantial relativity with their grounded, circumstantially relative representations, which in turn connect with action. I will call this the problem of **appropriately connected detachment**. Entirely *disconnected* detachment, as the surgery example shows, is likely to be easy enough to obtain (at least in some architectural sense), but on its own would not be significant. Totally *connected* detachment, though somewhat of a contradiction in terms, one be imagined as an explicit representation so locked into the default circumstances that it provides no power above and beyond what the grounded default case provided in the first place (tree rings might be an example—they are fully connected, at least for the live tree).

What is wanted is a mechanism that will continually mediate between the two kinds of representation—that will enable a system to shift, smoothly and flexibly, between indexical and implicit representations that can engender action, and generic and more explicit representations that enable it to communicate with others and in general have a certain detachment from its own circumstances. The problem, that is, is to provide something like an ability to "translate" between the two kinds (or, rather, among elements arranged along a continuum, or even throughout a space—as we have started to see, this is no simple dichotomy), just often enough to maintain the appropriate **causal connection** between located action and detached reasoning, but not so often as to lock them together. The right degree of partially causally connected self-reference, in other words, is our candidate for solving the problem of connected detachment. It enables a system to extricate itself from the limits of its own indexicality, and yet at the very same moment to remain causally connected to its own ability to act.

There is one final thing to be said about self-reference mechanisms in general, before turning to particular varieties. In any representational system, [it is widely agreed], the task domain or subject matter must be represented in terms of what we might call a *theory* or *conceptual scheme* that identifies the salient objects, properties, relations, etc., in terms of which the terms and claims of the representation are stated [i.e., in terms of which that task domain or subject matter is found intelligible]. With the possible exception of some extreme limiting cases, that is, [it is safe to say that] representation is *theory-relative*. By this I do not mean so much relative to an explicit account, in the sense of a theory viewed as a set of sentences, but relative to a way of carving the world up, a way of finding oneself coherent, a scheme of individuation.[h]

Granting this theory-relativity, we can see that causally connected self-reference requires the following three things:

1. **A theory of the self**, in terms of which the system's behaviour, structure, or significance can be found coherent. There is no *particular* aspect of the self that needs to be made explicit by this theory; we will see examples ranging from almost content-free sets of names, to complex accounts of internal properties and external relations.

[h] Point forward to the discourse on registration that I introduce—in O3?»

2. An **encoding of this theory within the system**, so that representations or impressions formulated in its terms can play a causal role in guiding the behaviour of the system.
3. A mechanism of **appropriate causal connection** that enables smooth shifting back and forth between direct thinking about, and acting in, the world, and detached reasoning about one's self and one's embedding circumstances. The only example we have seen so far is a mechanism that mediates between *k*-ary and *k*+1-ary representations of *n*-ary relations, as in the TO-THE-RIGHT-OF case; more complex examples will emerge.

The first two alone are not sufficient because they do not address the problem of causal connection. Thus the so-called "meta-circular interpreters" of List, as presented for example in Steele & Sussman (1978), meet the first two requirements, but since there is no connection between such meta-circular interpreters and the underlying system they are disconnected models of, they fail to meet the third. As such, they fail to meet the criterion of being able to serve as appropriately causally connected self-reference.

3 The Structure of Circumstance

I said earlier that particular mechanisms of self-reference can be understood as responses to different aspects of circumstantial relativity, which depend in turn on different aspects of circumstance itself. This means that, in order to understand these different mechanisms, we need an account of how circumstance is structured. This is a problem, for several reasons. First, there is probably no more problematic area of semantics. Second, we need a general account, since the whole point is to unify different proposals; nothing would be served by an account of how circumstance is treated by, say, semantic net impressions of a first-order language. Third, we *especially* cannot assume the circumstantial structure of traditional first-order logic, since the whole attempt to make logical and mathematical language "eternal" can be viewed as an attempt to rid such systems of as much circumstantial relativity as possible. Although that goal has not entirely been met, as the Carroll paradoxes show, the formulae of logical systems certainly lack some of the important kinds of relativity that characterise embedded systems.

Given these difficulties, my strategy will be to give a rough sketch of some of the possible structure of circumstance. All that I will ask is that what I provide support the demands of the next section. Since my basic aim is to show *how* the structure of self-reference reflects the structure of circumstantial relativity, any particular analysis of circumstance—including this one—can be taken as somewhat of an example.

By the **immediate** aspects or properties of a representational structure or impression l will mean those properties that can play a direct causal role in engendering any computational regimen defined over them. As such, they must not be relational—especially not to distal objects—but instead be locally and directly determinable (at least local and determinable within the system as a single whole), in such a way that a process interacting with or using the representation can "read off" [the presence or absence of an instantiation of] the property without further ado (i.e., without inference). Immediate aspects or properties, that is, must be immediately causally effective, in the sense that processes interacting with the structures can act differentially depending on their presence or absence—depending on whether or not they are occasioned.

For example, the (type) identity of tokens of a representational code (i.e., whether or not a given structure is a token of the word 'elaborate'), how many elements a composite structure has, etc., would on this account be counted as immediate. Non-immediate properties would include truth, being my favourite representation, and whether there is another type-identical representation elsewhere in a larger composite structure or system of which this particular representational structure is a part. This last example suggests that immediacy, which otherwise sounds like Fodor's notion of a *formal* property, is more locally restrictive, since all "internal" properties of a computational system, it seems, count as formal to him.[9] Positive existence will count as immediate, but negative existence not, since there is nothing for the latter property to be an immediate property of.

[9] Immediacy can also be less restrictive than formality, however, since I will countenance some semantic properties as immediate, such as the reference of direct quotations, small arithmetic properties exemplified by immediate structures, etc. See Fodor (1980) and Smith «forthcoming (a)».

Although it is tempting to compare the notion of an *immediate* property with apparently more familiar notions, such as of a *syntactic*, *intrinsic*, or *non-relational* property, such comparisons would involve us in more complexity than they are worth. The important point is merely that, with the notion of immediacy, I mean to get at those aspects of a representational structure that [are available to] affect or engender processes that use it; just what such potentially effective properties *are*, especially in any given case, is less important.[i]

In the last section I distinguished a system as a whole, its ingredient structures, and those structure's aspects or parts. With (i) that set of distinctions, (ii) our semantic notions of meaning, content, and significance, and (iii) the current notion of immediacy, we have in hand everything we are going to need to lay out the account of self-reference.

Specifically, I will say that something is **explicitly represented** by a structure or impression if it is represented by an immediate aspect of that structure. In contrast, something is **implicit** (with respect to an action or representation) if it is part of the circumstances that determine the content or significance of the representation or action, but is not explicitly represented. For example, I am explicitly represented by the sentence "I am now writing section 3 of this paper," since 'I' is a grammatical constituent of that sentence, and constituent identity is immediate. On the other hand, if I continue by saying "but I should stop because it is after midnight," and the word 'midnight' represents the time in the Pacific Time Zone, then the Pacific Time Zone is an implicit part of the relevant circumstances (even though it is not part of the reference of 'midnight'—i.e., of the metaphysical moment thereby referred to). Similarly, if I say "There is a bear to the right," I am implicitly involved, but not explicitly represented.

[i] To put this on the verge of pedantically, one could say that immediacy is a relational higher-order property, since it has to do with the ability of (a tokening or occasioning of) another property to cause an effect; whereas *syntactic*, *intrinsic*, *formal*, etc., could at least be argued to be non-relational higher-order properties, if one felt that whether a property was or was not a syntactic property depending solely on, as it were, 'local' or intrinsic facts about that property itself.

There are shades of a use/mention distinction in the way I am characterising the implicit/explicit distinction: things are explicitly represented (nothing, yet, is explicit on its own) only if they are out there in the content, so to speak—part of the described situation, or referents. Something is explicitly represented, that is, only if it is *mentioned*,[j] whereas something can be implicit either if it is used, or if it plays a middle role, not part of the sign itself, nor of the content or significance, but of the surrounding circumstances that mediate between the two. Thus the words of an utterance, on this view, are an implicit part of the circumstances that determine that utterance's content, since they are not themselves explicitly represented by the utterance (i.e., I am explicitly represented by the sentence "I am writing," but in that sentence the word 'I' plays only an implicit role). Where it will not cause confusion, however, I will also talk about *explicit or implicit representations of things*, as shorthand for "representations that represent those things explicitly or implicitly."

Finally, by extension, I will say that something is **explicit** *(simpliciter)* only if it meets two criteria: (i) it is explicitly represented, and (ii) it plays the role it plays in virtue of that explicit representation. So someone would be said to be an explicit part of a conversation only if they were explicitly referred to, and had whatever influence they had in virtue of that explicit representation. From this definition it follows that to **make something explicit** is to represent it explicitly in a causally connected way. Being implicit and explicit thus end up rather on a par, in the sense that both have to do with playing a role: to be *implicit* is to play a role directly; to be *explicit* is to play a role in virtue of being explicitly represented—which is to say, being represented by an immediate property.

We need to define one further notion, and then we are done. I

[j] Here and elsewhere throughout my writings, it is my habit to generalise the familiar notions of 'use' and 'mention' by extending 'mention' to apply to those objects referred to or named by *uses* of ground-level terms. Thus I would not only say (i) that, in the sentence "The word 'Nile' contains four letters," the *six*-character expression '*Nile*' is *used*, whereas the four-letter expression *Nile* is thereby mentioned (using italics in these last two phrases as a mentioning device!); but also (ii) that in the sentence "The Nile is more than four thousand miles long," the four-letter expression *Nile* is mentioned, and *a very long river is mentioned*.

have already called representational structures *self-relative* if different occurrences of them (or things of which those occurrences are a part) are part of the circumstances that determine their content. As pointed out above, however, there is more than one notion of part: part of the whole, and part of part of the whole. Rather than proliferating a raft of different mereological notions of self-relativity, it will be convenient merely to separate the facts and situations of the overall circumstances into three broad categories: **external circumstances**, having to do with parts of the world in which the overall system is not a participant; **indexical circumstances**, including those situations in the world at large in which the system is a constituent, and **internal circumstances**, including both the ingredient impressions, processes defined over them, relations among them, etc. Thus who is President, at the time of any given utterance or act of reasoning, and whether Shakespeare wrote the sonnet discovered in the Bodleian Library, would be paradigmatically external. Where a person or reasoning agent was, and whom it was talking to, would be (for it) indexical. Internal circumstances would include whether a represented formula's negation is also represented; what inference rules can be, or are being, applied; how often this impression has been used since the system's last cup of coffee; etc. Finally, *representations* will derivatively be called **external**, **indexical**, or **internal** (or a mixture) depending on whether their content depends on the corresponding kind of circumstance.

This typology allows us to say all sorts of natural things: that the agent plays an implicit role in the significance of THERE-IS-SOMETHING-TO-THE-RIGHT!; that 'I' is an explicit, indexical representation of an agent; that a truly unique identifier would be an explicit, non-indexical name; etc. Note also that a formula in a system of first order logic, at least in terms of its standard model-theoretic interpretation, has no implicit relativity to external or indexical circumstance (other than to the described situation itself), and no relativity to internal circumstance "outside" the formula, but aspects of it are nonetheless relative to the (implicit) internal structure of the formula itself. Whether an occurrence of variable is free, for example, or what quantifier binds it, is implicitly determined by the structure of the expression containing it. Prolog impressions, however, are implicitly relative to internal circumstances of the beyond-formula variety (because of such operations as CUT, etc.), and are often used indexically. For example, the Prolog term RIGHT(JOHN,MARY), if it meant that Mary was to the right of John *from the system's perspective*, would be counted as indexical.

4 Varieties of Self-Reference

We are now finally in a position to show how various mechanisms of self-reference facilitate various forms of connected detachment.

4a. Autonymy

I will call a system **autonymic** just in case it is capable of using a name for itself in an appropriately causally connected way. Just using a name that refers to itself does not make a system autonymic, even if that use affects the system in some way. What matters is that the name connect up, for the system, with its underlying, grounded, indexical architecture. To see this, imagine an expert system designed to diagnose possible hardware faults based on statistical analyses of reports of recoverable errors. Such a system might be given the data on its own recoverable errors, filed under a name known by its users to refer to it. The system's running this particular data set, furthermore, might eventually affect its very own existence (leading to board replacement, say). Even so, the system's behaviour in this case would not be any different from its behaviour in any other; it would yield up its conclusions entirely unaffected by the self-referential character of this externally provided name. When systems or agents respond differentially, however as for example do most electronic mail systems, which recognise and deal specially with messages addressed to their own users, forwarding other messages along to neighbouring machines—they will merit the autonymical label.

As we have already seen, two ingredients are required for autonymy. The first is a mechanism to convert between *k*-ary and *k*+1-ary impressions of *n*-ary relations.[10] For example, from the 0-

[10] For reasons that will be obvious, I do not think there is ever any reason—or need—to presume there is a final "fact of the matter" regarding how many arguments relations *really* have (or even that relations, as opposed to representations of them. *have* an "arity"). What is needed (for example in a scientific account) is a representation that makes explicit enough of the ar-

ary HUNGRY! and unary RIGHT(SOMEONE), we need to produce HUNGRY(__), and RIGHT(SOMEONE,__). Second, we need a term or name to use so that the new, more explicit, version has the same content as the prior, implicit version. This is required because, on the story we are telling, it is this particular explicit version that, in virtue of being connected, through the processes of causal connection, to the implicit perceptual and action-engendering version, gives any more general explicit versions their semantic integrity.

As the mail example suggests, something like a unique identifier can play this role. This is common in computational cases: designers of autonymic systems typically provide a way in which each system, though initially cast from the same mold, can be individually modified to react to its own unique name before being brought into service (a chore the system operators would do in "initializing" the system). As Perry suggests, however, this is not efficient: it requires that each system be structured somewhat differently. What is distinctive about the pronoun 'I', in contrast, is that it gives exactly (type-)identical systems a way of explicitly referring to themselves. 'I', in other words, is an indexical term allowing *explicit but self-relative* (hence efficient) self-reference. On its own it does not help a system escape from its indexicality, but, because it makes that indexicality explicit, it is the minimal step away from fully implicit indexicality.

Causal connections to implement autonymy are so simple as to seem trivial, but their importance outstrips their simple structure. The mail systems provide a good example: that each mail host recognise its own name, and attach its own name to messages headed out into the external world, is a simple enough task, but absolutely crucial to the functioning of the electronic mail community.

4b Introspection

In virtue of the inherent simplicity of names, purely autonymic mechanisms are almost completely theory-neutral. By **introspec-**

guments so as to be able to convey, as widely as possible, insight, understanding, truth, whatever. If the universe were in fact an ordered progression of big bangs, numbered 1–…, with k spatial dimensions and forces proportional to $1/r^{k-1}$ in each case (i.e., we are currently in the third round), all the relations of physics would turn out to have another parameter. That would be OK.

tive systems, in contrast, I will refer to systems with causally connected self-referential mechanisms that render explicit, in some substantial way, some of their otherwise implicit internal structure. Since most of the self-referential mechanisms that have actually been proposed fall in this class, this variety of self-reference will occupy most of our remaining attention.

The first step, in analysing introspective systems, is to distinguish our own theoretical commitments from the theoretical commitments we attribute to the agents we study. The difference can be seen by comparing Levesque's logic of "explicit" and "implicit" belief[11] (his terms, not ours, though the meanings are similar) with Fagin & Halpern's logics of belief and awareness.[12] Levesque's use of the predicates *B* and *L* for explicit and implicit belief are predicates of the theorist: nothing in his account—as he himself notes—commits him to the view that the agents he describes parse the world in terms of anything like the belief predicate (i.e., in Fagin & Halpern's phrase, they need not be "aware" of the belief predicate). Fagin and Halpern, on the other hand, when they use such axioms as Bf ⇒ BBf, thereby commit the agents to an awareness of the same belief predicate they themselves use, I.e., for us to say "*A* believes f" is for us to adopt the notion of belief; for us to say "A believes *that it believes* f" commits *A* to the notion of belief as well. Iterated epistemic axioms such as Bf ⇒ BBf can therefore be substantially misleading, since any inner (non-initial) B's must represents the agents' notion; the outer ones will be only the theorists'.

In the self-referential models typical of the autoepistemic tradition, the correspondence between explicit representation and belief is so close that this identification of agent's and theorist's commitment seems harmless, but when we deal with more complex introspective theories we will have to allocate theoretical commitments more carefully. For example, some theories that are straightforward, from a theorist's point of view, may be difficult or impossible for introspective systems to use, if they assume a perspective necessarily external to the agents they are theories of. Furthermore, different introspective theories require different primitive

[11] Levesque (1984).

[12] Fagin & Halpern (1985)

("wired-in") support, whereas we, as external theorists, can use any theory we like, without fear of architectural consequence. For example, it is only a small move for a theorist to change from a theory of a programming language that objectifies only the environment, to one that also objectifies the continuation. On the other hand, programming systems that can introspect using continuations are an order of magnitude more subtle than ones that introspect solely in terms of environments (we will see why this is so in a moment).

Keeping these cautions in mind, consider, as a first introspective example, an almost trivial autoepistemic computational agent comprising a set of base level representations, whose content, though perhaps self-relative, has primarily to do with facts about the world external to the system. As is usual in such cases, we will presume that the *representation* of each fact, within the system, engenders the system's belief in that fact—that is, we will adopt the *Knowledge Representation Hypothesis* laid out in Smith (1985)—so for familiarity we will call these representations *beliefs* rather than impressions. Ignore reasoning entirely, for the moment, and assume that the agent believes only what has somehow been stored in its memory. For introspective capability, augment the base set of beliefs with a set of sentences formulated in terms of what Levesque calls an *explicit belief predicate*. So, for example, as well as containing the "belief" MARRIED(JOHN), imagine the system also being able to represent B(MARNLED(JOHN)).[13] I will call the whole system S, and its simple introspective representations *B*-sentences. (Note: In this and subsequent discussion [am representing impressions *within* S, not giving theoretical statements in an external logic *about* S, so sentences of the form f represent beliefs S already has, and *B*-sentences represent introspective beliefs. All occurrences of *B*, in other words, represent theoretical commitments on S's part.)

S's *B*-sentences, though introspective, are still implicit and indexical, in several ways. First, the agent doing the believing—i.e., S itself—remains implicitly (and efficiently) determined by internal circumstance, as does the current belief set with respect to which

[13] Or, if you prefer, B('MARRIED(JOHN)'). For purposes of this paper I do not need to take a stand on the question of the semantic or syntactic nature of believe objects—which is fortunate, because I no longer think it is a well-formed question. See «Smith forthcoming (b)».

the *B*-sentence derives its truth conditions. I.e., *B*(a) is true just in case a is one of the base-level sentences, meaning that it is explicitly represented in S's general internal store, which will presumably change over time. Furthermore, by hypothesis, any implicitness or indexicality of S's base-level beliefs is inherited by the *B*-sentences: *B*(RIGHT(*x*)) is no more explicit about RIGHT's other three arguments than is the simpler RIGHT(*x*).

Given that S is so simple, do the *B*-sentences do any useful work? Since we have claimed that introspective representations render explicit what was otherwise implicit, it is natural to wonder what otherwise implicit aspect of S's base-level beliefs these *B*-sentences represent. The answer requires a simple typology of "relations of structured correspondence". In particular, I will call a representation **iconic** (what is sometimes called *analogue)* if it represents each object, property, and relation in the represented domain with a corresponding object, property, and relation in the representation (iconic representations are thus fully explicit). Similarly, I will say that a representation **objectifies** any property or relation that it represents with an object.[k] Thus for example the sentence MARRIEO(JOHN,MARY) objectifies marriage, since it uses (an instance of) the object 'MARRIED' to signify (an instance of) the relation of marriage that connects John and Mary. A representation **absorbs** any object, property, or relation that it represents with itself (thus the grammar rule EXP ⇒ OP(EXP,EXP) absorbs left-to-right adjacency). Finally, I will say that a representation is **polar** just in case it represents an absence with a presence, or vice versa (positive polarity in the first case, negative in the second). For example, the absence of a key in a hotel mail slot is often taken to signify the presence of the tenant in the hotel, making mail slots a negatively polar iconic representation of occupancy.[l]

If all *B*-sentences were positive, then S's introspective representations would be a partial, non-polar, iconic representation of its base level beliefs (partial because we are not necessarily assuming *B*(a) for all a). Since such representations objectify nothing, and therefore do not increase the explicitness of the base level, they are

[k] «These notions of iconicity, objectification, absorption, and polarity are taken from "The Correspondence Continuum,", q.v.»

[l] Needless to say, an example from the 1980s.

not of much use on their own. Causal connection for them is also relatively trivial. Negative B-sentences, however, of the form $\neg B$(a), make the introspective representations positively polar, thereby objectifying an otherwise implicit property of base level representations: namely, the property of negative existence (we have already seen that negative existence is not immediate, which forces it to be implicit, unless explicitly represented, as in this case). Thus $\neg B$(a) makes explicit one of the simplest imaginable implicit properties of a set of internal representations. No slight on importance is suggested, but it is noteworthy how close the correspondence between introspective impression and baseqevel impression remains: the objects of the introspective level correspond one-to-one with the objects of the base level: only a single, unary property is objectified (no relations); etc. Nonetheless, as logicians are not the only ones who know, that one act of "rendering something explicit" can have substantial computational consequences, because—once appropriate causal connection is provided—it makes immediate what was not otherwise immediate, with the effect that computational consequence can depend directly on the absence of a belief, which it could not (at least not easily) do in the non-introspective version.

Causal connection, even with the positive polarity, is still relatively simple. B(a) will be true just in case a is an element of the set of representational impressions, and although negative existence is not an immediate property of the belief set, constituent identity in a finite set is, so that negative existence can be "computed" with only a moderate amount of inference—just a membership check on the base level belief set. Thus returning 'yes' or 'no' upon being *asked* "B(a)?" is relatively straightforward. It is less clear what should happen if $\neg B$(a) were to be *asserted,* although one can easily imagine a system in which this would either trigger a complaint, if a were already in the base set of impressions, or else perhaps cause its removal.

This example illustrates what will become an increasingly common theme: whether causal connection is typically easy or hard depending on two things:

1. The explicitness of the introspective representation (that is, the closeness of correspondence between the immediate

properties of the introspective representation and its content); and

2. The immediacy of the aspects of self thereby explicitly represented.[m]

An explicit representation of immediate properties of base-level beliefs, that is (such as their "syntactic" properties, their presence or absence, which we have in this case, etc.), sustains relatively straightforward causal connection.[14] This equation—immediacy on both ends, simply connected—is hardly surprising, since immediacy is what engenders computational effect, and computational effect is required at both ends of causal connection. To the extent that either (i) immediacy on either end is lessened, or (ii) the connection between them becomes more complex, causal connection typically becomes that much more difficult.

Examples of such difficulty are not hard to come by. They arise as soon as we complicate the example and consider introspective impressions that represent more complex internal properties—particularly relational ones. Curiously, in these more realistic cases introspective relativity itself tends to rise, as well as the non-immediacy of what is represented. Thus consider Moore's (1983) interpretation of M(a) as "a is consistent." This introspective representation is locally indexical because it is relative to the entire base-level set of representations, which is not explicitly represented with its own parameter. Moore himself points out this relativity:

> "The operator M changes its meaning with context just as do indexical words in natural language, such as 'I', 'here', and 'now'… Whereas default reasoning is nonmonotonic because it is defeasible, autoepistemic reasoning is nonmonotonic because it is indexical. "[15]

As it happens, however, this indexicality is not what makes the causal connectivity of consistency difficult; rather, the problem

[m] «Check this out—is this really right? In particular, isn't it correspondence between immediacy and immediacy? Is that what explicitness comes to?

[14] This is really the point made in Konolige (1985).

[15] Moore (1983) pp. 6–7. By 'meaning' Moore means what we are here calling content, and by 'indexical' he means what we mean by 'internally relative,' but his point of course is valid.

stems from the fact that property of consistency is *not itself immediate*, but a (computationally expensive) relational property of the entire base-level set. Similarly, when interpreted as "implied (or entailed) by the base level set," as in both Konolige and Fagin & Halpern,[16] *B* becomes a relational, not immediate property (though again it is circumstantially relative), and causal connection consequently grows problematic.

The environment and continuation aspects of the control structure of Lisp programs, made explicit in the introspective 3Lisp,[17] are also implicit, but not relational, and therefore more computationally tractable than consistency. 3Lisp is so designed that causal connection is supported in both directions (see below); as well as obtaining a representation of what the continuation was, you can also cause the continuation to be *as represented*. So in 3Lisp you can *assert* the introspective representation (it is not clear what that would mean under the consistency reading of *M*(a), for example). Similarly, various different aspects of the Prolog proof procedure—goal set, control strategy, output—are made introspectively explicit in Bowen & Kowalski's amalgamated logic programming proposals.[18] Again, the consistent assumption sets in a truth-maintenance system, typically implicit, are made explicit in deKleer's assumption-based truth maintenance system ATMS.[19]

Since it would be hopeless to delve into these or other introspective proposals in depth, I will devote the remainder of this section to three broad problems they all must deal with. Before doing so, however, it is important to note that the introspective models that typify the autoepistemic tradition represent an extremely constrained conception of introspective possibility. Admittedly, that tradition does not limit introspective beliefs to *B*(a) or ¬*B*(a), with *B* meaning "is immediately represented in the base level set," as our initial example suggests; the consistency reading of *M*, as Moore's example shows, and readings of *B* (or *L*) as "is implied by the rest of the belief set" are much more complex, as the discussion of causal connection makes clear. Nonetheless, such accounts can still largely be viewed as positively polar, iconic representations of

[16] «ibid, ibid»

[17] «Ref»

[18] «Ref»

[19] deKleer (1986).

derivable extensions of the base set. There is no inherent reason, however, to limit introspective deliberations to such one- or two-predicate vocabularies: one can easily imagine systems with introspective access to proof mechanisms and the state of proof procedures (as is typical in proposals from the control camp), or theories of self that deal with whether ground-level beliefs are chauvinist, creative, or largely derived from children's books. The kinds of meta-level reasoning that prompted Artificial Intelligence's original interest in self, cited for example in Collins (1975), are not limited to knowing *what* one believes, but having some *understanding* of it. The potential subject matter of introspection, in other words, should be understand to be at least as broad as necessary to include clinical psychology and psychiatry, and perhaps sociology as well. In sum, whereas one can agree with Konolige's (1985) opening statement that "introspection is a general term covering the ability of an agent to reflect upon the workings of his own cognitive functions," there is no reason to limit those reflections as drastically as he does in constraining his "ideal introspective agents" to think nothing more interesting than "do I or don't I believe a?"

4.b.i *Introspective Integrity*

The three issues that must be faced by any model of introspection are largely independent of basic cognitive architecture or theory of self. The first l call **introspective integrity**: it includes all questions of whether introspective representations are true, but extends as well to questions of whether any other significant properties they have (truth is only one) mesh appropriately with their content. In S,'s case integrity is relatively simple: *B*(a) should be represented just in case a is, and ¬*B*(a) just in case a is not. This simplicity depends partly on the simplicity of the introspective representational language, but also on another property of S we have not yet mentioned: the truth of S's introspective structures depends only on facts about the base-level representations, independent of introspective commentary. For an example where this does not hold, imagine a system where any impression (base-level or otherwise) is believed *unless there is introspective annotation stating otherwise*. Such a system would probably profit from an explicit representation of the truth and belief predicates, so that statements like "I should probably believe this, even though Mary doubts it," and "This cannot be true, because it conflicts with something else I believe" could be straightforwardly represented (truth-maintenance systems are not unlike this). In such a case it would be natural to ask of any given base-level impression whether it is believed, but this cannot be settled by inspecting only the base-level impressions. It would depend both on the state of the base level memory *and on implications of the introspective commentary*, and might therefore be arbitrarily difficult to decide. The truth-functional integrity of such a system would thus be inextricably relational.

Integrity is not offered as a property an introspective system must achieve, but rather as a notion with which to categorise and understand particular introspective axioms and mechanisms. For example, all of Konolige's notions of "ideality," "faithfulness," and "fulfillment" can be viewed as proposals for kinds of partial integrity. Similarly, Fagin and Halpern's $A_i\mathrm{f} \Rightarrow A_i A_i\mathrm{f}$ axiom for self-reflective systems is an axiom that ensures introspective integrity for their notion of awareness. In a particular case even outright introspective falsehoods could be licensed.

Truth is not the only significant property, and therefore is not the only aspect of integrity that matters, as we can see by looking at Bowen and Kowalski's DEMO predicate.[20] According to the standard story, logic programs have both a declarative reading, under which clauses can be taken as formulae in a first-order language, and a procedural reading, under which they (implicitly) specify a particular control sequence, which implements a particular instance of the proof (derivability) relation. It follows that the *declarative* reading of DEMO should signify an abstraction over the (implicit) *procedural* regimen (i.e., [[DEMO]] = £, to be a little cavalier about notation). But this is not all that is required, if DEMO is to play the role that Bowen and Kowalski imagine; it must also be the case that the *procedural* reading of DEMO—i.e., the control sequence engendered by an instance of DEMO(PROG,GOALS)—must also lead to GOALS' being (actively) derived from PROG. Similarly, in 3Lisp, where 'f' was used in the external theory to signify content (i.e., roughly [[...]]), and 'c" to signify procedural consequence (roughly, £), and where the internal (impression) designing pro-

[20] «Ref»

cedural consequence was called NORMALISE was the internal impression representing procedural consequence,[21] it was necessary to show not only that f(NORMALISE)=c, but also, very roughly (ignoring some use/mention issues) that c(NORMALISE)≈c. The general point is the following: suppose you have an impression A of some aspect P of the internal state (i.e., such that $[\![A]\!]=P$). In order for this to count as having *rendered P explicit* (rather than just as representing P explicitly!), a use of this representation A of P must also *engender P* (remember, we said that something is rendered explicit only if it subsequently participates in the circumstances in virtue of that representation).

Intuitively, what this all comes to is something like the following. In order to count as having introspective access to some aspect of your self, not only must you be able to *represent* that aspect; you must also be able to *use* that representation—to step through it, so to speak, in what we informally call "problem-solving mode"—in such a way that this introspective deliberations *can serve as one way of doing what is being introspected about.* At this level of generality, the characterisation should not be contentious—though in some cases it might seem like a luxury, since after all there are things we can think about (such as how we ride a bicycle) that we cannot simulate in virtue of reasoning with those thoughts. But one of the advertised powers of introspection is its ability to enable us to do things differently from how our underlying architecture would have done them, had we not introspected. Moreover, cognitive introspection is *thinking about thinking,* two instances of the same type of activity—as opposed to thinking about bicycling, where the thinking and the bicycling are at least in some sense rather different.[22] And if a system cannot at least think or reason (introspectively) in the same way (modulo timing) that it would have had it not done so introspectively, there seems little chance that it will ever be able to move beyond its base level capabilities. This is part of what causal connection demands. Thus, according

[21] I.e., c=NORMALISE, as it were, in which the term on the left is in the theorist's external analytic language, and the term on the right is in 3Lisp's internal language.

[22] Which is not to deny that bicycling "without thinking" may well land one in danger.

to our account, although I can think about how I ride a bicycle, since I cannot ride a bicycle by thinking about it, my bicycle-riding thoughts do not qualify for the label *causally-connected introspection*.

4.b.ii *Introspective Force*

The second major issue, once again having to do with causal connection, is what I call **introspective force**. It has to do not with the causal efficacy of the introspective structures themselves, but with the causal connection between those structures and the aspects of self they represent. This is the problem addressed by what in the literature have been called *linking rules*, *reflection principles*, *semantic attachment*, *level-shifting*, etc.,[23] although simple quotation and disquotation operators are even simpler examples—e.g., Interlisp 's KWOTE and (some of its uses of) EVAL; 3Lisp's ↑ and ↓, etc. In the discussion so far, I have characterised causal connection rather symmetrically, as a relation between representations and actual aspects of self. As the sophistication of introspection increases, however, the relation between self and self-representation not only grows more complex, but the two directions of connection—from self to representation (I will call this "upwards"), and from representation to self ("downwards")—take on rather different properties. The differences are at least analogous to (what current ideology takes as) the distinction between beliefs and goals.

Imagine, to borrow an example from Smith (1984), paddling a canoe through whitewater, exiting an eddy leaning upstream (the wrong thing to do), and taking a dunking. If, sitting on the bank a few moments later, you were to think about how to do better, you would first have to obtain an explicit representation of what you were doing just a moment earlier (this is the "belief" case: how do you go from a fact to a true belief about it?). It is no good to think "Ah, yes, the second millennium is drawing to a close," as it was when you fell in; you want to represent the very local situation that led you to fall into the river, represented in the appropriate

[23] 'Linking rule' is used in Bowen & Kowalski (1982), 'semantic attachment' in Weyhrauch (1980), 'level-shifting' in des Rivi6res and Smith (1984), and 'reflection principles' in Weyhrauch (1980) and some of the meta-logical tradition.

way. This is the connection from reality (i.e., self) to representation. But similarly, after analysing the affair, and concluding that things would have gone better if you had leaned the other way, you do not want merely to sit on the bank, fatuously contemplating an improved self: the idea is to get back in the water and do better. That is, you need a connection from representation to reality (more like the situation when you have a goal or even intention): you have a representation, and you want the facts to fit it). Both kinds of connection are germane even for as simple a self-referential representation as $\neg B$(a); the system might need to know whether $\neg B$(a) is true, or it might want to make it true. On S's reading of *B* as "is explicitly represented" neither direction is too hard: if *B* means "consistent," the story, as we have already noted, would be very different.

As McDermott and Doyle (1980) discovered, it is easy to motivate perfectly determinate readings for introspective predicates where the causal connection is not computable, even upwards.[x] In the downwards case, moreover, if the property represented is a relational one, there may be no unique determinate solution (lots of things, typically, could make $\neg M$(a) true). It is thus a substantial problem, in actually designing an effective introspective architecture, to put in place sufficient mechanism to mediate between general introspectively represented goals and the specific actions on the self that have the dual properties of being causally connected (so that they can be put into effect) and satisfying the goal in question.

Since this problem is simply a particular case of the general issue of designing and planning action, however, and not specific to the introspective case, it need not concern us more here.

4.b.iii Introspective Overlap

The third issue that must be faced by introspective systems is what I will call the problem of **introspective overlap**, which arises when the implicit circumstances of introspective impression coincide with, or include, what has been rendered explicit. The issue arises because the introspective representations are themselves part of what constitutes the agent. As such, any claims they make that involve, explicitly or implicitly, properties of the whole state of the

[x] Tai!! Is this what I want to say? Is it what they said?

agent, will be claims that they are likely, in virtue of their own existence or treatment, to affect (but not effect!). Introspective representations of relational properties that obtain between a particular impression and the whole set are obvious candidates for this difficulty. For example, if six beliefs were represented, one could not truthfully add the impression

TOTAL-NUMBER-OF-EXPLICITLY-REPRESENTED-BELIEFS(6)

Instead, one would need to add

TOTAL--OF-EXPLICITLY-REPRESENTED-BELIEFS(7)

This overlap between content and circumstance is what opens the way for the puzzles and paradoxes of narrow self-reference. It is a more general notion than strict "‘circularity," since the problems can arise even if the representational structure itself is not part of its own content. An early but familiar example in computer science arose in the case of debugging systems for programming languages with substantial interpreter state, when written in the same language as the programs they were used to debug. These debugging systems, introspective by our account, rendered explicit the otherwise implicit parts of the control state of some other fragment of the overall system. The problem was that they too engendered control state (used global variables, occupied stack space, etc.), thereby introducing a variety of confusions because of unwanted conflict. These confusions often occasioned extraordinarily intricate code to sidestep the most serious problems, sometimes with only partial success. The fundamental problem, however, is easily described in our present terminology: overall, the implicit dimension or aspect of the system that was rendered explicit remained the implicit dimension or aspect of the explicit rendering. There was no circularity involved, but there was overlap, with concomitant problems.

Overlap is not necessarily a mistake: the indexicality that ‘I’ renders explicit is the same indexicality that implicitly gives the pronoun its content (similarly for ‘here’ and ‘now’). Problems seem to arise only when negatives or activity affect what would otherwise be the case. It is typically necessary, in such cases, to give an introspective mechanism an appropriate *vantage point* or *layered set of implicit contexts,* analogous to that provided by type hierarchies in logic, so that the introspective process can muck about with its subject matter without affecting the circumstances that give that subject matter its content.

Overlap only arises when the introspective machinery makes explicit some implicit aspect of the internal circumstances; it is not a problem when what is implicit to the base-level is also implicit for the introspective machinery. Thus various systems, such as MRS and Soar,[24] apparently do not make explicit any otherwise implicit state (everything that can be seen, self-referentially, is *already* explicit; what is implicit remains so), so the problem of overlap does not arise. In some other cases, such as in BROWN,[25] overlap would occur, but the power of the introspective machinery is curtailed in advance to avoid contradiction. Handling overlap coherently was one of the problems that 3Lisp was designed to solve: its purpose was to demonstrate the compatibility, in a theory-relative introspective procedural system, of detached vantage point, substantial implicit state, and complete causal connection.[26] The continuation structures of 3Lisp, representing the dynamic state of the overlapping processor, were what made it interesting. The other two aspects that were made explicit—structural identity, roughly, and lexical environment—did not overlap (this is why, as we said earlier, an introspective variant of 3Lisp that only rendered these two aspects explicit would be essentially trivial).

3Lisp's particular solution to the problem of overlap was to provide what amounted to a type hierarchy for control, and in terms of that to provide, as a primitive part of the underlying architecture, mechanisms that always maintained the integrity of the connection between self-representation and facts thereby represented. Such a tight connection was made possible in 3Lisp—because, as stated, continuations are not relational—that its actual (and perfectly effective) behaviour could be demonstrated to be equivalent, in an important sense, to that that would have been manifested by the infinite idealisation in which all of its internal aspects (relative to its highly constrained theory) were always explicitly repre-

[24] «Refs»

[25] Friedman and Wand (1984)

[26] At the time of its design I called 3Lisp 'reflective,' not 'introspective,' but I now think this was a mistake. Reflection—see below—was what I wanted, but introspection was what I succeeded in providing.

sented to itself. As a consequence, both external theorist and internal program could pretend, even with respect to recursively specified higher ranks of introspection, that it was indefinitely introspective with perfect causal connection. This particular architecture, however, will clearly not generalise to more comprehensive introspective theories, such as those involving consistency.

There is obviously no limit to the expressiveness of introspective representation, or intricacy of causal connection, although there are very real limits on the total combination of introspective expressiveness, integrity, and force. In the human case it seems clear that causal connection is the practical problem, especially in the "downwards" direction—from representation to fact: though it is not exactly easy to come by accurate psychological self-knowledge, it seems much harder, given such knowledge, to become the person you can so easily represent yourself to be.

The real challenge to self-reference, however, stems not from the limits on introspection, where after all one has, at least in some sense, access to everything being theorised about, but from the difficulty of obtaining a non-indexical representation of one's participation in the external world.

4c. Reflection

In the last section a point was made that we need to go back to, because within it lie the seeds of the limits of introspective self-reference. In particular, it was pointed out, in connection with the move from the base-level RIGHT(x) to the introspective B(RIGHT(x)), that all of the implicitness of the former is inherited by the latter. The self-relativity of the single-argument RIGHT—the fact that three of its four arguments get filled in by the indexical circumstances of the agent—is left implicit even in the introspective version. By a **reflective** system, in contrast, I will mean any system that is not only introspective, but that is also able to represent the external world, including its own self and circumstances, in such a way as to *render explicit*, among other things, *the indexicality of its own embeddedness*. This representational capacity, however, is (as usual) insufficient on its own; the system must at the same time retain causal connection between this detached representation, and its basic, indexical, non-explicit representations, which enable it to act in that external world.

Like substantial introspection, reflection is thus something we can only approximate; complete detachment is presumably impossible, both because no one knows to what extent properties that seem universal are in fact local but just happen to hold throughout our limited experience, and because it is very likely, for reasons of efficiency, that we will not ever have represented them. Reflection is also hard to attain, because of the requirement of causal connection. Finally, in order to obtain a representation of oneself that is truly external—i.e., that would hold from an external agent's perspective—one must first represent to oneself everything implicit about one's internal structure and state that is not universally shared (or anyone shared by one's peers). Without this kind of self-knowledge, what one takes to be a detached representation of the world will still be implicitly self-relative, in ways one presumably will not realise. Introspection is therefore a prerequisite for substantial reflection (self-knowledge is a precursor of detachment, as history has repeatedly told us). Yet in spite of these difficulties, reflection is necessary if one is to escape from the confines of self-relativity.

What then can we say about reflection, if it is so important? No very much—at least yet. Of the three self-referential traditions we have been tracking, neither the autoepistemic nor the control has addressed relativity to the external world at all. In both cases the self-referential focus has remained internal, though for different reasons. In the autoepistemic case, the "language" typically used for external representation either has either been, or has been closely based on, mathematical logic—which, as Barwise and Perry have repeatedly emphasized, does not admit, in its foundations, of external relativity to circumstance. Hence logic's focus on *sentences*, rather than on *statements*, and its semantic models of *mathematical structures*, not *situations in the world*. In spite of all this, however, as pointed out earlier, even purely mathematical systems are permeated with internal implicitness: with questions of consistency, truth, etc. It is this internal relativity on which autoepistemic models of self-reference have therefore concentrated.

The control tradition stems more directly from computer science and programming language semantics, which have by and large trafficked in internal accounts. Its failure to deal with external relativity is roughly the dual of the autoepistemic's: whereas the autoepistemic tradition has dealt with external *content*, but not with *external relativity*, computer science has focused on *complex relativity*, but not on the *external world*. Hence computer science's self-referential tradition—the control camp—has also dealt only with internal introspection. Programs, in particular, are typically viewed as (procedural) specifications of how a system should behave; as a result their subject matter is taken to be the internal world of the resulting system: its structures, operations, behaviour.[n] Although one can (and I do) argue that the resulting computational systems are themselves representational, and therefore bear a "content" relation to the world in which they are ultimately deployed, that system-world relation is not addressed by traditional programming language analyses. As a result, the implicitness represented by such self-referential models as meta-circular interpreters, BROWN, MRS, etc.,[27] is also primarily internal.[28]

Thus there is somewhat of a gap between the self-referential mechanisms that have so far been proposed (which are primarily introspective), and the accounts of external relativity offered by the circumstantial camp. What we need are mechanisms for rendering that external implicitness explicit. As usual, causal connection will be the difficult problem—more difficult than for introspection, since internal circumstance, to the extent that it is causally effective at all, is always within the causal reach of the agent. The consistency of a set of first-order sentences may be difficult or impossible for a formal system to ascertain, but that is not because

[n] «Point (forwards?) towards the "ingredient" vs. "specificational" view.

[27] See Steele & Sussman (1978), Friedman and Wand (1984), and Genesereth et al. (1983), respectively.

[28] Not realising this fully at the time, I did not initially describe 3Lisp (Smith 1982, 1984) in a way that was very accessible to the programming language community. 3Lisp's semantical model, in particular, was based on a conception of computation where the subject matter of a program was taken to include not only the system whose behaviour was being engendered, but also the subject matter of the resulting system. I still believe that this is often how programming is *understood*, even if implicitly, by a large number of programmers: my analysis; however it would have been more accessible had this non-standard semantic conception been treated more explicitly. Ironically, however, in spite of this semantical orientation, the only "external" world 3Lisp was able to deal with was that of pure (and simple) mathematics, so it did not really live up to its own semantical mandate.

there is crucial information somehow beyond the reach of that system, remote in time and space, to which other systems might have better access. Determining consistency is hard *all by itself.* The external circumstantial dependencies of ordinary language and thinking, however, are different: who is the right person to perform some particular function, for example, is something that only the world can ever know for sure. The best reflective agent will have direct causal access—and probably only partial access at that—to only one potential candidate.

None of this means that serious reflection is impossible, however, partly because of our three-way, rather than two-way, categorisation of circumstance into external, indexical, and internal types. The truth of whether Shakespeare wrote the sonnet is external; the implicitness motivated by efficiency, in contrast, is typically indexical, not external, and indexicality has to do with the circumstances in which the agent participates—which circumstances, some of which, at least, should be relatively *nearby.* If there is any locality in this world, there seems more hope of an agent's knowing about local circumstances than about situations arbitrarily remote in space and time. What is enduringly difficult, of course, is that even those circumstances must be represented as if by another.

5 The Limits of Self-Reference

Perfect self-knowledge is obviously impossible, for at least three reasons: (i) because of the complexity of the calculations involved, such as those illustrated by consistency; (ii) because of the theory-relativity—no theory can render *everything* explicit; and (iii) because some circumstantial relativity—particularly indexical and external—remains beyond the causal reach of the agent. But there are other limits as well, An important one stems from the fact that the self being represented is ultimately the same self as the one doing the representing, and as such certain possibilities are physically (if not metaphysically) excluded. The self can never be viewed in its entirety, because there is no place to stand—no vantage point from which to look.

Another limit—more a danger than a constraint—was intimated at the outset: although introspection (and self-knowledge) is a prerequisite to substantial reflection, it remains true that the power of all of these mechanisms derives ultimately from their ability to support more general, more detached, more communicable reasoning. It is a danger, however, that in climbing up out of its embedded position, a system will end up thinking solely about its self, rather than using its self to get outside itself. This would lead to a self-involved—ultimately autistic—sort of system, of no use whatsoever.

These limits notwithstanding, self-reference and self-understanding are important. One can look out, see three people around the table, and represent the situation with "there are four people at this dinner party." One may also notice, perhaps with only introspective capability, that one is repeating oneself. But then one goes on to observe that, by doing so, one is acting inappropriately: that from the other three's perspective one looks like a fool. And then—here is where causal connection gets its bite—as soon as one has achieved this detached view of the situation, this representation from the outside, one scurries back into the introspective state, replaces the designator of that fourth person with 'I', recognises its special self-referential role, collapses back down to the fully implicit structures that engender talking, cuts them off, and thereby shuts up.

That is almost as good as writing more briefly.

Acknowledgements

I am indebted to everyone involved in AI's inquiry into self-reference, for their participation in what I take to be a collaborative inquiry. Particular recognition goes to Jim des Rivières, fellow traveler in the upper reaches of 3Lisp, and to all other members of the Knights of the Lambda Calculus.[p] Finally, a special debt is owed to John Perry: although most of the analysis in section 4 precedes his influence, much of the framework in which it is presented here, especially the emphasis on indexical relativity, has benefited from his works. My thanks.

References

Barwise, Jon, and Perry, John (1983): *Situations and Attitudes*, Cambridge, MA: Bradford Books.

[p] «This organization should probably be explained ;-).»

Batali, John (1983): "Computational Introspection", MIT Artificial Intelligence Laboratory Memo AIM-701. Cambridge Mass.

Bowen, Kenneth A., and Kowalski, Robert A. (1982): "Amalgamating Language and Metalanguage in Logic Programming", in K. L. Clark and S.-A Tariund (eds.), *Logic Programming*, New York: Academic Press.

Collins, A. M., Warnock, E., Aiello, N, and Miller, M (1975): "Reasoning from Incomplete Knowledge", in Bobrow, D. G., and Collins, A. (eds.), *Representation and Understanding*, New York: Academic Press.

Davis, Randall (1976): "'Applications of meta level knowledge to the construction, maintenance, and use of large knowledge bases", Stanford AI Memo 283 (July 1976), reprinted in Davis, R., and Lenta, D. B. (eds.) *Knowledge-Based Systems in Artificial Intelligence*, New York: McGraw-Hill.

——— (1980): "Meta-Rules: Reasoning About Control", *Artificial Intelligence* 15: 3, pp. 179–222.

de Kleer, John, Doyle, Jon, Steele, Guy L., and Sussman, Gerry J. (1979): "Explicit Control of Reasoning", in P. H. Winston and R. H. Brown (eds.), *Artificial Intelligence: An MIT Perspective*, Cambridge: MIT Press.

de Kleer, Johan (1986): "An Assumption-Based TMS", *Artificial Intelligence*, to appear in 1986.

des Rivières, James and Smith, Brian Cantwell (1984): "The Implementation of Procedurally Reflective Languages", *Proc, Conference on LISP and Functional Programming*, pp. 331–47, Austin Texas. Also available as Xerox PARC Intelligent Systems Laboratory Technical Report ISL-4, Palo Alto, California, 1984.

Doyle, Jon (1980): "A Model for Deliberation, Action, and Introspection", MIT Artificial Intelligence Laboratory Memo AIM-TR-581, Cambridge Mass.

Fagin, Ronald, and Halpern, Joseph Y. (1985): "Belief. Awareness, and Limited Reasoning: Preliminary Report," *Proceedings IJCAI-85* pp. 491-501, Los Angeles, California.

Fodor, Jerry (1980): "Methodological Solipsism Considered as a Research Strategy in Cognitive Psychology," *The Behavioural and Brain Sciences*, 3: l, pp. 63–73. Reprinted in Fodor, J., *RePresentations*, Cambridge, MA: Bradford, 1981.

Friedman. Daniel P., and Wand, Mitchell (1984): "'Reification: Reflection without Metaphysics," *Proc. Conference on LISP and Functional Programming*, pp. 348–55, Austin Texas.

Genesereth, Michael R., and Smith, David E. (1982): "Meta-Level Architecture," Stanford Heuristic Programming Project Technical Report HPP-8I-6, version of December 1982, Stanford California.

Genesereth, Michael R., Greiner, Richard. and Smith, David E. (1983): "MRS—A Meta-Level Representation System," Stanford Heuristic Programming Project Technical Report HPP-83-27, Stanford California.

Halpern, Joseph Y., and Moses, Yoram (1985): "A Guide to the Modal Logics of Knowledge and Belief, Preliminary Draft," *Proceedings of IJCAI-85*, pp. 480–90, Los Angeles, California.

Harman, Gilbert (1982): "Conceptual Role Semantics," *Notre Dame Journal of Formal Logic*, 23, pp. 242–56.

Hayes, Patrick J. (1973): "Computation and Deduction," *Proceedings of the 1973 Mathematical Foundations of Computer Science* (MFCS) *Symposium*, Czechoslovakian Academy of Sciences.

Kaplan, David (1979): "On the Logic of Demonstratives," in P. A. French, T. E. Uehling. Jr., and H. K. Wettstein, (eds.), *Perspectives in the Philosophy of Language*, Minneapolis, pp 383–412.

Konolige, Kurt (1985): "A Computational Theory of Belief Introspection," *Proceedings of* IJCAI-85. pp. 502–08, Los Angeles, California.

Kowalski, Robert (1979): "'Algorithm=Logic+Control," CACM **22**, pp. 424–36.

Laird, John E., and Newell, Allen (1983): "A Universal Weak Method: Summary of Results," in *Proceedings of* IJCAI-83, pp. 771–73. Karlsruhe, West Germany.

Laird. John E., Newell, Allen, and Rosenbloom, Paul S. (forthcoming): "SOAR: An Architecture for General Intelligence", forthcoming.

Lenat, Douglas B., and Brown, John Seely (1984): "Why AM and EURISKO Appear to Work," *Artificial Intelligence* **23**, pp. 269–94.

Levesque, Hector J. (1984): "A Logic of Implicit and Explicit Belief," *Proceedings of the AAAI-84 Conference*, pp. 198–202, Austin. Texas. A revised and expanded version available as FLAIR Technical Report 32, Fairchild Artificial Intelligence Laboratory, Palo Alto, California, 1984.

McDermott, Drew, and Doyle, Jon (1980): "Non-Monotonic Logic I," *Artificial Intelligence* 13:1&2, pp. 41–72.

Moore, Robert C. (1983): "'Semantical Considerations on Nonmonotonic Logic," Artificial Intelligence Center Technical Note 284, SRI International, Menlo Park, California.

Perlis, Donald (1985): "Languages with Self-Reference I: Foundations," *Artificial Intelligence* 25, pp. 301–22.

Perry, John (1983): "Unburdening the Self," unpublished manuscript, presented at the Conference on Individualism, Center for the Humanities, Stanford University, Stanford California.

——— (1985a): "Self-Knowledge and Self-Representation," in *Proceedings of IJCAI-85*, pp. 1238–42, Los Angeles, California.

——— (1985b): "Perception, Action, and the Structure of Believing," in Grandy & Warner (eds.): *Philosophical Grounds of Rationality*, Oxford: Oxford University Press, pp. 330–59.

——— (forthcoming): "Thought Without Representation," to be presented at a Joint Symposium of the Mind Association and the Aristotelian Society, London, July 1986.

Rosenschein, Stanley J. (1985): "Formal Theories of Knowledge in AI and Robotics," in David Nitzan, ed., *Proceedings of Workshop on Intelligent Robots: Achievements and Issues,* SRI International, Menlo Park, California.

Smith, Brian Cantwell (1982): *Reflection and Semantics in a Procedural Language*, M.I.T. Laboratory for Computer Science Technical Report MIT-TR-272.

——— (1984): "Reflection and Semantics in Lisp," *Conference Record of 11th Principles of Programming Languages Conference* (POPL), pp. 23–35, Salt Lake City, Utah. Also available as Xerox PARC Intelligent Systems Laboratory Technical Report ISL-5, Palo Alto, California, 1984.

——— (1985), "Prologue to *Reflection and Semantics in a Procedural Language*," reprinted in R. Brachman and H. Levesque (eds.), *Readings in Knowledge Representation,* Los Altos, CA: Morgan Kaufman, pp. 31–39.

——— (forthcoming a): "Is Computation Formal?," Stanford University CSLI Technical Report.

——— (forthcoming b): "Categories of Correspondence", Stanford CSLI Technical Report.

Steele, Guy L. Jr, and Sussman. Gerry J. (1978): "The Art of the Interpreter; or, the Modularity Complex (Parts Zero, One and Two)," M.I.T. Artificial Intelligence Laboratory Memo No 453, Cambridge, MA.

Weyhrauch, Richard W. (1980): "Prolegomena to a Theory of Mechanized Formal Reasoning," *Artificial Intelligence* **13**: l&2, pp. 133–70.

— Were this page been blank, that would have been unintentional —

C • Computing

— Were this page been blank, that would have been unintentional —

6 — The Limits of Correctness[†]

Abstract

There is a formal technique in computer science, known as **program verification**, which is used, in its own terms, to "prove programs correct". From its name, someone might easily conclude that a program that had been proven correct would never make any mistakes, or that it would always follow its designers intentions. In fact, however, what are called *proofs of correctness* are really *proofs of the relative consistency between two formal specifications*: one of the program, one of the model in terms of which the program is formulated. Part of assessing the correctness of a computer system, however, involves assessing the appropriateness

[†] Lightly edited version of a paper originally presented at a *Symposium on Unintentional Nuclear War* at the *Fifth Congress of the International Physicians for the Prevention Nuclear War*, Budapest, Hungary, June 28–July 1, 1985. The version delivered there was subsequently reprinted: (i) as Center for the Study of Language and Information Report CSLI–85–36, Stanford, California: Stanford University, Oct. 1985, 22 pp.; (ii) in D. Johnson & H. Nissenbaum (eds.), *Computers, Ethics & Social Values*, Englewood Cliffs, NJ: Prentice Hall, 456–69; (iii) in Colburn, T. R., Fetzer, J. H., & Rankin T. L. (eds.), *Program Verification*, Kluwer Academic Publishers, Dordrecht/Boston/London, 1993, pp. 275–93; and (iv) in Kling, R. (ed.), *Computerization and Controversy: Value Conflicts and Social Choices* (2nd Ed.), San Diego: Academic Press, pp. 810–25. This version differs from those in having some grammatical infelicities cleared up, dated references explained, plus some mild copy-editing for purposes of clarity.

Support gratefully acknowledged: in the 1980s, from the Xerox Corporation and from the System Development Foundation through an award to the Center for the Study of Language and Information (CSLI) at Stanford University; and in 2009 from the Jackman Humanities Institute, University of Toronto.

of this model. Whereas standard semantical techniques are relevant to the program-model relationship, we do not currently have any theories of the further relationship between the model and the world in which the program is embedded.

In this paper I sketch the role of models in computer systems, comment on various properties of the model-world relationship, and suggest that the term 'correctness' (in the program verification context) should be changed to 'consistency.' In addition I argue that, since models cannot in general capture all the infinite richness of real-world domains, complete correctness is inherently unattainable, for people or for computers.

1 Introduction

On October 5, 1960, the American Ballistic Missile Early Warning System station at Thule, Greenland, indicated a large contingent of Soviet missiles headed towards the United States.[1] Fortunately, common sense prevailed at the informal threat-assessment conference that was immediately convened: international tensions were not particularly high at the time, the system had only recently been installed—and perhaps most salient of all, Soviet Premier Khrushchev happened to be in New York. All in all, a massive Soviet attack at that particular moment seemed very unlikely. And so no devastating counterattack was launched.

What was the problem? The moon had risen, and was reflecting radar signals back to earth. Needless to say, this lunar reflection had not been predicted by the system's designers.

Over the last few decades, the United States Defense Department has spent many millions of dollars on a computer technology known as "**program verification**"—a branch of computer science whose business, in its own terms, is to "prove programs correct". Program verification has been studied in theoretical computer science departments since a few seminal papers in the 1960s,[2] but it was only in the late 1970s and 1980s that it started

[1] Edmund Berkeley, *The Computer Revolution*, Doubleday, 1962, pp. 175–77, citing newspaper stories in the *Manchester Guardian Weekly* of Dec. 1, 1960, a UPI dispatch published in the *Boston Traveler* of Dec. 13, 1960, and an AP dispatch published in the *New York Times* on Dec 23, 1960.

[2] McCarthy, John, "A Basis for a Mathematical Theory of Computation,"

to gain in public visibility, and to be applied to real world problems. General Electric, to consider just one example, initiated verification projects in their own laboratories, hoping to prove that the programs used in their computer-controlled washing machines would not have any "bugs" (even a single serious one in a major product can destroy a corporation's profit margin).[3]

Although it used to be that only the simplest programs could be "proven correct"—programs to put simple lists into order, to compute simple arithmetic functions, etc.—slow but steady progress has been made in extending the range of verification techniques. By the early 1980s papers began to report correctness proofs for somewhat more complex programs, including small operating systems, compilers, and other materiel of modern system design.[4]

What, we do well to ask, does this new technology mean? How good are we at it? For example, if the 1960 warning system had been proven correct (which it was not), could we the problem with the moon have been avoided? If it were possible to prove that programs written to control automatic launch-on-warning systems were correct, would that provide us with assurance that there will not—and could not—be a catastrophic accident? In systems currently being designed computers will make counterattack launch decisions in a matter of seconds, with no time for any human intervention (let alone for musings about Khrushchev's being in New York). Do the techniques of program verification hold enough promise that, if these new systems could all be proven correct, we could all sleep more easily at night?

These are the questions I want to look at in this paper. And my answer, to give away the punch-line, is *no*. For fundamental

1963, in P. Braffort and D. Hirschberg, eds., *Computer Programming and Formal Systems*, Amsterdam: North-Holland, 1967, pp. 33–70. Floyd, Robert, "Assigning Meaning to Programs," *Proceedings of Symposia in Applied Mathematics* 19, 1967 (also in F. T. Schwartz, ed., *Mathematical Aspects of Computer Science*, Providence: American Mathematical Society, 1967). Naur, P., "Proof of Algorithms by General Snapshots," BIT Vol. 6 No. 4, pp. 310–16, 1966.

[3] Albert Stevens, BBN Technologies, Inc. (called "Bolt, Beranek and Newman" at the time), personal communication.

[4] See for example R. S. Boyer, and Moore, J S., eds., *The Correctness Problem in Computer Science*, London: Academic Press, 1981.

reasons—reasons that anyone can understand, and that no one can escape—there are inherent limitations to what can be proven about computers and computer programs. Although program verification is an important new technology—useful, like so many other things, in its particular time and place—it should definitely not be called *verification*. Just because a program is "proven correct", in other words, you cannot be sure that it will do what you intend.

First some background.

2 General Issues in Program Verification

Computation has become the most important enabling technology of nuclear weapons systems: it underlies virtually every aspect of the defense system, from the early warning systems, battle management and simulation systems, and systems for communication and control, to the intricate guidance systems that direct the missiles to their targets. It is difficult, in assessing the chances of an accidental nuclear war, to imagine a more important question to ask than whether these pervasive computer systems will or do work correctly.

Because the subject is so large, however, I want to focus on just one aspect of computers relevant to their correctness: the use of **models** in the construction, use, and analysis of computer systems. I have chosen to look at modelling because I think it exerts the most profound and, in the end, most important influence on the systems we build. But it is only one of an enormous number of important questions. First, therefore—in order to unsettle you a little—let me just hint at some of the equally important issues I will *not* address:

1. **Complexity:** At the current state of the art, only very simple programs can be proven correct. Although it is terribly misleading to assume that either the complexity or power of a computer program is a linear function of length, some rough numbers are illustrative. The simplest possible arithmetic programs are measured in tens of lines; the current state of the verification art extends only to programs of up to several hundred. It is estimated that the systems proposed in the Strategic Defense Initiative (Stars Wars),

in contrast, will require at least ten billion (10,000,000) lines of code.[5] This is a difference of at least five decimal orders of magnitude. By analogy, compare the difference between resolving a two-person dispute and settling the political problems of the Middle East. There is no a priori reason to believe that strategies successful at one level will scale to the other.

2. **Human interaction:** Not much can be "proven," let alone specified formally, about actual human behaviour. The sorts of programs that have so far been proven correct, therefore, do not include much substantial human interaction. As the moon-rise example indicates, on the other hand, it is often crucial, in the design of complex systems, to allow enough human intervention to enable people to override system mistakes and cope with unanticipated eventualities. System designers, therefore, are faced with a very real dilemma: (i) should they rule out substantive human intervention, in order to develop more confidence in how their systems will perform; or (ii) should they include it, so that costly errors can be avoided or at least repaired? The partial core meltdown at the Three Mile Island generation plant in 1979 is a trenchant example of just how serious this tradeoff can get: the system design provided for considerable human intervention, but in the event the operators failed to act "appropriately." Which strategy leads to the more important kind of correctness?

   A standard way out of this dilemma is to specify the behaviour of the system *relative to the actions of its operators.* But as we will see below, this strategy pressures the designers to specify the system totally in terms of internal actions, not external effects. So the best that a proof can end up demonstrating is that the system will *behave in the way that it will behave* (i.e., it will raise this line level 3 volts), not that it will *do what you want it to do* (i.e., launch a mis-

[5] Fletcher, James C., study chairman, and McMillan, Brockway, panel chairman, *Report of the Study on Eliminating the Threat Posed by Nuclear Ballistic Missiles, Vol. 5, Battle Management, Communications. and Data Processing,* U. S. Department of Defense, February 1984.

sile only if the attack is real). Unfortunately, the latter is clearly what is important. Systems comprising computers and people must function properly as integrated systems; nothing is gained by showing that one cog in a misshapen wheel is a very nice cog indeed.

Furthermore, large computer systems are dynamic, constantly changing, embedded in complex social settings. Another famous "mistake" in the American defense system occurred when a human operator mistakenly mounted a training tape, containing a "simulation" of a full-scale Soviet attack, onto a computer that, just by chance, was automatically pulled into service when the primary machine ran into a problem. For some tense moments the simulation data were taken to be the real thing.[6] What does it mean to install a "correct" module into a complex social flux?

3. **Levels of Failure:** Complex computer systems must work at many different levels. It follows that they can *fail* at many different levels too. By analogy, consider the many different ways a hospital could fail. First, the beams used to frame it might collapse. Or they might perform flawlessly, but the operating room door might be too small to let in a hospital bed (in which case you would blame the architects, not the lumber or steel company). Or the operating room might be fine, but the hospital might be located in the middle of the woods, where no one could get to it (in which case you would blame the planners). Or the hospital, in spite of having been "properly built", might have been damaged by an unanticipated (and unanticipatable) earthquake. Or, to take a different example, consider how a letter could fail. It might be so torn or soiled that it could not be read. Or it might look beautiful, but be full of spelling mistakes. Or it might have perfect grammar, but

[6] See, for example, the Hart-Goldwater report to the Committee on Armed Services of the U.S. Senate: "Recent False Alerts from the Nation's Missile Attack Warning System" (Washington, D.C.: U.S. Government Printing Office, Oct. 9, 1980); Physicians for Social Responsibility, *Newsletter*, "Accidental Nuclear War," (Winter 1982), p. 1.

disastrous contents.

Computer systems are the same: they can be "correct" at one level—say, in terms of hardware—but fail at another (i.e., the systems built on top of the hardware can do the wrong thing even if the chips are fine). Sometimes, when people talk about computers failing, they seem to think that it is only the hardware that needs to work properly. Sure enough, hardware does from time to time fail, causing machines to come to a halt, or yielding errant behaviour (as for example when a faulty chip in another American early warning system sputtered random digits into a signal interpreted as indicating how many Soviet missiles had been sighted, again causing a false alert[7]). And the connections between the computers and the world can break. On the day in which when the moonrise problem was recognized, an attempt to override it failed because an iceberg had accidentally cut an undersea telephone cable.[8]

The more important point is that, in order to be reliable, a system must be correct, or anyway reliable, *at every relevant level*. The hardware is just the starting place—and by far the easiest, at that. Unfortunately, however, we do not even know what all the relevant levels are. So-called "fault-tolerant" computers, for example, are particularly good at coping with hardware failures, but the software that runs on them is not thereby improved.[9]

4. **Correctness and Intention:** What does *correct* mean, anyway? Suppose the people want peace, and the President thinks that means having a strong defense, and the Defense department thinks that having a strong defense requires maintaining an arsenal of nuclear weapons systems, and the weapons designers request control systems to monitor radar signals, resulting in computer companies

[7] Ibid.

[8] Berkeley, op. cit. See also Daniel Ford's two-part article "The Button," *New Yorker*, April 1, 1985, p. 43, and April 8, 1985, p. 49, excerpted from Ford, Daniel, *The Button*, New York: Simon and Schuster, 1985.

[9] Developing software for fault-tolerant systems is an extremely tricky business.

> being asked to develop systems that respond to six particular kinds of radar pattern, and the engineers are told to build signal amplifiers with certain circuit characteristics, and the technician is told to write a program to respond to the difference between a two-volt and a four-volt signal on a particular incoming wire. If being correct means *doing what was intended*, whose intent matters? The technician's? Or what, with twenty years of historical detachment, we would say *should have been intended?*

With a little thought any of you could extend this list yourself. And none of these issues even touch on the intricate technical problems that arise in developing mathematical analyses of the software and systems used in the so-called "correctness" proofs. But, as I said, I want to focus on what I take to be the most important issue underlying all of these concerns: the pervasive use of **task domain models**. Models are ubiquitous not only in computer science but also in human thinking and language; their very familiarity makes them hard to appreciate. So we will start simply, looking at modelling on its own, and come back to correctness in a moment.

3 The Permeating Use of Models

When you design and build a computer system, you first—wittingly or unwittingly—formulate a model of the problem you want it to solve, and then construct the computer program in its terms. For example, if you were to design a medical system to administer drug therapy, you would need to model a variety of things: the patient, the drug, the absorption rate, the desired balance between therapy and toxicity, and so on and so forth. The absorption rate might be modelled as a number proportional to the patient's weight, or proportional to body surface area, or as some more complex function of weight, age, and sex.

Similarly, computers that control traffic lights are based on some model of traffic—of how long it takes to drive across the intersection, of how much metal cars contain (the signal change mechanisms are triggered by wires buried under each street). Bicyclists, as it happens, often have problems with automatic traffic lights, because bicycles do not exactly fit the model: they do not contain enough iron to trigger the metal detectors. I also once saw a tractor get into trouble because it could not move as fast as the system "thought" it would: the light allowing cross-traffic to enter the intersection went green when the tractor was only half-way through.

To build a model is to conceive of the world in a certain delimited way. To some extent you must build models before building any artifact at all, including televisions and toasters, but computers have a special dependence on these models: to write a *program* is effectively to *write down an explicit description of the model inside the computer*, in the form of a set of rules or what are called *representations*—essentially a set of linguistic formulae encoding, in the terms of the model, the facts and data thought to be relevant to the system's behaviour. It is with respect to these representations that computer systems work. In fact that is really what computers are (and how they differ from other machines): they run by manipulating representations, and representations are always formulated in terms of models. This can all be summarized in a slogan:[10]

No computation without representation.

The models, on which the representations are based, come in all shapes and sizes. Balsa models of cars and airplanes, for example,

[10] Footnote added 2009: It is no longer considered necessary for programs to represent the structure of the task domains in which they work—especially to represent it *explicitly*, in a set of language-like formulae or expressions. A great deal of "situated artificial intelligence," the use of network "models" in dynamic-systems based software, etc., the development of machine learning, etc., which has taken place over the twenty-five years since this paper was written, can be understood as various kinds of attempt exactly to avoid such explicit task domain representation. However: (i) it remains overwhelmingly likely that any software system designed and built to control a major military system of the sort being discussed would still be built on top of an explicit model—if for no other reason than that this design strategy allows the model to be updated, if and as appropriate, when the systems involved change (e.g., the nature and number of missiles, sensors, etc.), without having to build the entire code base over again; and (ii) even machine learning networks and connectionist systems and the like rely on models (some even develop their own)—it is just that the *representation* of the model in the system may be less explicit that was taken for granted twenty-five years ago.

are used to study air friction and lift. Blueprints can be viewed as models of buildings; musical scores as models of a symphony. But models can also be abstract. Mathematical models, in particular, are so widely used that it is hard to think of anything that they have not been used for: from whole social and economic systems, to personality traits in teenagers, to genetic structures, to the mass and charge of sub-atomic particles. These models, furthermore, permeate all discussion and communication. Every expression of language can be viewed as resting implicitly on some model of—some assumed conceptual structure or "take" on—the world.

What is important for our purposes is that every model deals with its subject matter *at some particular level of abstraction*, paying attention to certain details, throwing away others, grouping together similar aspects into common categories, and so forth. So the drug model mentioned above would probably pay attention to the patients' weights, but ignore their tastes in music. Mathematical models of traffic typically ignore the idiosyncratic temperaments of individual taxi drivers. Sometimes what is ignored is set aside because it is considered to be at too "low" a level to matter, for the system's ultimate purpose; sometimes it is ignored because it is too "high": it all depends on the purposes for which the model is being used. So a hospital blueprint would pay attention to the structure and connection of its beams, but not to the arrangements of proteins in the wood the beams are made of (too low), nor to the efficacy of the resulting operating room (too high).

Models *must* ignore things exactly because they view the world at a level of abstraction.[11] And it is good that they do: otherwise they would drown in the infinite richness of the embedding world. Though this is not the place for metaphysics, it would not be too much to say that every act of conceptualization, analysis, categorization, does a certain amount of *violence* to its subject matter, in order to get at the underlying regularities that group things together. If you do not commit that act of violence—if you do not ignore some of what is going on—you would become so

[11] 'Abstraction' derives from the Latin '*abstrahere*'—to pull or draw away.

hypersensitive and so overcome with complexity that, as a finite creature, you end up paralyzed, unable to act.

To capture all this in a word, I will say that models are **inherently partial**. All thinking, and all computation, are similarly partial. Furthermore—and this is the important point—thinking and computation *must* be partial: that is how they are able to work.

4 Full-blooded Action

Something that is not partial, in contrast, is **action**. When you reach out your hand and grasp a plow, it is the *real field* you are digging up, not your model of it. When you talk to someone of a different race or culture, what you say may be inexorably affected by your model of the person, their community, their social group, etc.—but your addressee is an *actual person*—not your or anyone else's model of that person.[12]

Models, in other words, may abstract (may "pull away"), and thinking may abstract, and some aspects of computation may abstract—but *action does not*.[13] To actually build a hospital, to clench the steering wheel and drive through the intersection, to inject a drug into a person's body—to do any of these things is to act in the full-blooded world, not in a partial or distilled model of it.

This difference between action and modelling is extraordinarily important. To move from thought or intent to concrete action is to take leave of one's model and participate in the whole, rich, infinitely variegated world. For this reason, among others, action plays a crucial role, especially in the human case, in grounding the more abstract processes of modelling or conceptualization. One form that grounding can take, which computer systems can already take advantage of, is to provide feedback on how well the modelling is going. For example, if an industrial robot develops an internal three-dimensional representation of a wheel assembly

[12] This is not to deny that we are *affected by*—perhaps even (partially) *constituted by*—our own, our families', and our society's models of us. What it denies is that we *are* such models.

[13] Even if what action it is is affected or determined by the actor's model of the situation in which the action is undertaken.

passing by on a conveyor belt, and then guides its arm towards that object and tries to pick it up, it can use video systems or force sensors to see how well the model corresponded to what was actually the case. The world does not care about the model: the claws will settle on the wheel just in case the actualities mesh.

Feedback is a special case of a very general phenomenon: you often learn, when you do act, just how good or bad your conceptual model was. You learn, that is, if you have adequate sensory apparatus, the capacity to assess the sensed experience, the inner resources to revise and reconceptualize, and the luxury of recovering from minor mistakes and failures.

5 Computers and Models

What does all this have to do with computers, and with correctness? The point is that computers, like people, and unlike mathematics, are *engaged participants* in the world. Like us, they *participate* in the real world: they take real actions; they cause effects, and are affected by causes. One of the most important facts about computers, to put this another way, is that they are concrete; they use energy; we plug them in. They are not, as some theoreticians seem to suppose, pure mathematical abstractions, living in a pure detached heaven—or detached simulacra or "models in themselves," living a hermetically sealed life in a parallel universe. On the contrary, computers land real planes at real airports; administer real drugs; and—as we know only too well—control real radars, missiles, and command systems. Like us, in other words, although they base their actions on models, they have consequence in a world that inevitably transcends the partiality of their enabling models. Like us, in other words, and unlike the objects of mathematics, they are challenged by the inexorable conflict between partial but tractable models and actual but infinite reality.

And, to make the only too obvious point: we in general have no guarantee that the models are right—indeed we have no *guarantee* about much of anything about the relationship between model and world. As we will see, current notions of "correctness" do not even address this fundamental question.

In philosophy and logic, as it happens, there is a very precise

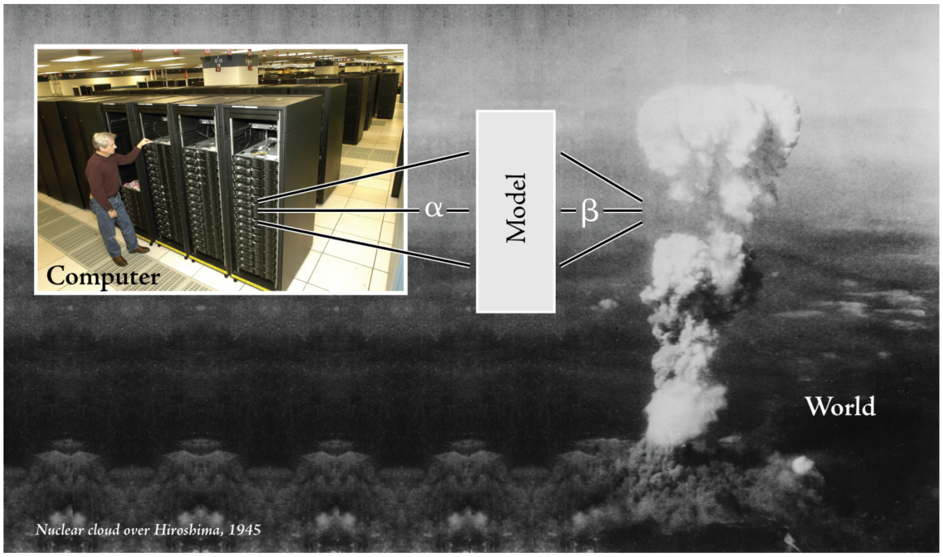


Figure 1 — Computers, Models, and the Embedding World

mathematical theory called "model theory." You might think that it would be a theory about what models are, what they are good for, how they correspond to the worlds they are models of, and so forth. You might even hope this was true, for the following reason: a great deal of theoretical computer science, and all of the work in program verification and correctness, historically derives from this model-theoretic tradition, and depends on its techniques. Unfortunately, however, model theory does not address the model-world relationship at all. Rather, what model theory does is to tell us how our descriptions, representations, and programs *correspond to our models*.

The situation, in other words, is roughly as depicted in Figure 1. You are to imagine a description, program, computer system (or even a thought—they are all similar in this regard) in the left hand box, and the very real world in the surrounding right. Mediating between the two is the inevitable model, serving as an idealized or pre-conceptualized simulacrum of the world, in terms of which the description or program or whatever can be understood. One way to understand the model is as the glasses through which the program or computer looks at the world: it is the world, that is, *as the system sees it* (though not, of course, as it necessarily *is*).

The technical subject of "model theory," as I have already said, is a study of the relationship, labeled a, on the left. What about relationship b, on the right? The answer, and one of the main points I hope you will take away from this discussion, is that, at this point in intellectual history, *we have no theory of this right-hand side relationship—no theory of the relationship between models and the world.*

There are lots of reasons for this lack— some very complex. For one thing, most of our currently accepted formal techniques were developed during the first half of this century to deal with mathematics and physics. Mathematics is unique, with respect to models, because (at least to a first level of approximation) its subject matter is the world of models and abstract structures, and therefore the model-world relationship is relatively unproblematic. The situation in physics is more complex, as is the relationship between mathematics and physics. How apparently pure mathematical structures can be so successfully used to model the material substrate of the universe is a question that has exercised physical scientists for centuries.[14] But the point is that, whether or not one believes that the best physical models do more justice and therefore less violence to the world than do models in so-called "higher-level" disciplines like sociology or economics, formal techniques do not themselves address the question of adequacy.

Another reason we do not have a theory of the right-hand side is that there is very little agreement on what such a theory would look like. In fact all kinds of question arise when one studies the

[14] Cf. Eugene Wigner's "The unreasonable effectiveness of mathematics in the natural sciences," *Communications in Pure and Applied Mathematics,* Vol. 13, No. I (February 1960). New York: John Wiley & Sons, Inc.

model-world relationship explicitly: about whether it ean be treated formally; whether it can be treated rigorously, even if not formally; what the relationship is between those two approaches, whether any theory will be more than usually infected with the prejudices and preconceptions of the theorist; and so forth. The investigation quickly leads to foundational questions in mathematics, philosophy, and language, as. well as computer science. But none of what one learns in any way lessens its ultimate importance. In the end, any adequate theory of action, and, consequently, any adequate theory of correctness, will have to take the model-world relationship into account.

6 Correctness and Relative Consistency

Let us get back, then, to computers, and to correctness. As I mentioned earlier, the word 'correct' is already problematic, especially as it relates to underlying intention. Is a program correct when it does what we have *instructed* it to do? or what we *wanted* it to do? or what history would dispassionately say it *should have done*? Analysing what correctness *should* mean is too complex a topic to take up directly. What I want to do, in the time remaining, is to describe what sorts of correctness we are presently capable of analysing.

In order to understand this, we need to understand one more thing about building computer systems. I have already said, when you design a computer system, that you first develop a model of the world, as indicated in Figure 1. But in general, you never get to hold the model in your hand. Computer systems, in general, are based on models that are purely abstract. Rather, if you are interested in proving your program "correct," you develop *two concrete things,* structured in terms of the abstract underlying model (although these are listed here in logical order, the program is very often written first):

1. A **specification**: a formal description in some standard formal language, specified in terms of the model, in which the desired behaviour is described; and

2. The **program**: a set of instructions and representations, also formulated in the terms of the model, which the computer uses as the basis for its actions.

How do these two differ? In various ways, of which one is particularly important. The program has to say *how the behaviour is to be achieved,* typically in a step-by-step fashion—often in excruciating detail. The specification, however, is less constrained: all it has to do is to specify *what proper behaviour would be,* independent of how it is accomplished.

A specification for a milk delivery system, for example, might simply be: "Make one milk delivery at each store, driving the shortest possible distance in total." That is an adequate description of what has to happen. The program, on the other hand, would have the much more difficult job of saying how this was to be accomplished. It might be phrased as follows: "Drive four blocks north, turn right, stop at Gregory's Grocery Store on the corner, drop off the milk, then drive 17 blocks north-east…". Specifications, to use some of the jargon of the field, are essentially *declarative;* they are like indicative sentences or claims. Programs, on the other hand, are *procedural:* they must contain instructions that lead to a determinate sequence of actions.

What, then, is a proof of correctness? It is a proof that any system that *obeys the program* will *satisfy the specification*.

There are, as is probably quite evident, two kinds of problems here. The first, often acknowledged, is that the correctness proof is in reality only a proof that two characterizations of something are compatible. When the two differ—i.e., when you try to prove correctness and fail—there is no more reason to believe that the first (the specification) is any more correct than the second (the program). As a matter of technical practice, specifications tend to be extraordinarily complex formal descriptions, just as subject to bugs and design errors and so forth as programs. In fact they are very much like programs, as this introduction should suggest. So what almost always happens, when you write a specification and a program, and try to show that they are compatible, is that you have to adjust both of them in order to get them to converge.

For example, suppose you write a program to factor a number *C*, producing two answers *A* and *B*. Your specification might be:

Given number C, produce numbers A and B such that $A \times B = C$

This is a specification, not a program, because it does not tell you how to come up with *A* and *B*; all it say is what properties *A* and *B* should have. In particular, suppose I say: "OK, *C* is 8,687,001,541; what are *A* and *B*? Staring at the specification just given will not help you to come up with an answer.[15] Suppose, on the other hand, given this specification, that you then write a program—say, by successively trying pairs of numbers until you find two that work. Suppose further that you then set out to prove that your program meets your specification. And, finally, suppose that this proof can be constructed (I will not go into details here; I trust you can imagine that such a proof could be constructed). With all three things in hand—program, specification, and proof—you might think you were done.

In fact, however, things are rarely that simple, as even this simple example can show. In particular, suppose, after doing all this work, that you try your program out on some simple examples, confident that it must work because you have a proof of its correctness. You randomly give it 14 as an input, expecting 2 and 7. But in fact it gives you the answers $A=1$ and $B=14$. In fact, you realise upon further examination, it will *always* give back $A=1$ and $B=C$. It does this, *even though you have a proof of its being correct*, because you did not make your specification meet your intentions. You wanted both *A* and *B* to be *different* from C (and also different from 1), but you forgot to say that. In this case you have to modify both the program and the specification. A plausible new version of the latter would be:

> *Given number C, produce numbers A and B such that* $A \neq 1$ *and* $B \neq 1$ *and* $A \times B = C$.

We still are not done. If the next version of the program, given $C=14$, produces $A=-1$ and $B=-14$, you would once again have met your new specification, but still failed to meet your intention, leading you to propose something like:

> *Given number C, produce numbers A and B such that* $A \neq 1$ *and* $A \neq -1$ *and* $B \neq 1$ *and* $B \neq -1$ *and* $A \times B = C$.

And so on. I take it that the point is obvious. Writing "good" specifications—which is to say, writing specifications that capture

[15] Probably what you had in mind were 84,719 and 102,539.

your intention—is *hard*.

It should be apparent, nonetheless, that developing even straightforward proofs of "correctness" is nonetheless very useful. As illustrated in this almost trivially simple example, doing so often forces you to delineate, very explicitly and completely, the model on which both program and specification are based—as well as to articulate, again very explicitly, your often tacit assumptions. A great many of the simple bugs that occur in programs, of which the problem of producing 1 and 14 was an example, arise from sloppiness and unclarity about the model. Such bugs are not identified, *per se*, by the proof, but they are often unearthed in the attempt to prove the equivalence. And of course there is nothing wrong with this practice; anything that helps to eradicate errors and increase confidence is to be applauded. The point, rather, is to show exactly what these proofs consist in.

In particular, as the discussion has shown, when you show that a program meets its specifications, all you have done is to show that two formal descriptions, slightly different in character, are compatible. This is why I think it is somewhere between misleading and immoral for computer scientists to call this "correctness". What is called a *proof of correctness* is really a proof of the compatibility or consistency between two formal objects of an extremely similar sort: program and specification. As a community, we computer scientists should call this **relative consistency**, and drop the word 'correctness' completely.

Even if rightly renamed, proofs of relative consistency still ignore the second problem intimated earlier. Nothing in the so-called program verification process per se deals with the right-hand side relationship: the relationship between the model and the world. But, as is clear, inadequacies on the right hand side—inadequacies, that is, in the models in terms of which the programs and specifications are written—remain common reasons for system failure.

The problem with the moon-rise was a problem of this second sort. The difficulty was not that the program failed, in terms of the model. Rather, the problem was that the model was overly simplistic; *it did not correspond to what was the case in the world.*

Or, to put it more carefully, since all models fail to correspond to the world in indefinitely many ways, as we have already said, it did not correspond to what was the case *in a crucial and relevant way*. In other words, to answer one of our original questions, *even if a formal specification had been written for the 1960 warning system*, and a proof of correctness generated, there is no reason to believe that potential difficulties with the moon would have emerged.

You might think that the designers were sloppy; that they would have thought of the moon if they had been more careful. But it turns out to be extremely difficult to develop realistic models of any but the most artificial situations, and to assess how adequate these models are. The example of factoring numbers brought some of this to the fore, but as another example, think back on the case of General Electric, and imagine writing appliance specifications, this time for a refrigerator. To give the example some force, imagine that you are contracting the manufacture of the refrigerator out to an independent supplier, and that you want to put a specification into the contract that is sufficiently precise to guarantee that you will be happy with anything that the supplier delivers that meets the contract.

Your first version might be quite simple—say, that the requisitioned device should maintain an internal temperature of between three and six degrees Centigrade; not use more than 200 watts of electricity; cost less than $100 to manufacture; have an internal volume of half a cubic meter; and so on and so forth. But of course there are hundreds of other properties that you implicitly rely on: it should, presumably, be structurally sound: you would not be happy with a deliciously cool plastic bag. It should not weigh more than a ton, or emit loud noises. It should not fling projectiles out at high speed when the door is opened. And so on—essentially *ad infinitum*. It is generally impossible, when writing specifications, to include *everything* that you want: legal contracts, and other humanly interpretable specifications, are always stated within a background of common sense, to cover the myriad unstated and unstatable assumptions assumed to hold in force. (Current computer, alas, have no common sense, as the cartoonists know so well—so they cannot be asked to interpret their programs against such a reasonable background.)

So it is hard to make sure that everything that meets your specification will really be a refrigerator; it is also hard to make sure that your requirements do not rule out perfectly good refrigerators. Suppose for example a customer plugs a toaster in, puts it inside the refrigerator, and complains that the object they received does not meet the temperature specification—and must therefore not be a refrigerator. Or suppose they try to run it upside down. Or complains that it does not work in outer space, even though you did not explicitly specify that it would only work within the earth's atmosphere. Or suppose they install it in an expensive centrifuge running at 100,000 rpm and discover that at that speed all the air is pushed up against one wall, again causing it not to work. Or suppose they just unplug it. These cases are the dual of the former—the problem is not that what is claimed to be a refrigerator is not one, but that what is in fact a refrigerator is claimed not to be one. And in each one of them, you would say that the problem lies not with the *refrigerator* but with the *use*. But how is *use* to be specified?

A constitutive part of modelling an artifact, in other words, involves understanding the relevant part of the world in which it will be embedded, and the relevant ways it will be used. One could try to extend the notion of modeling to cover that, too—i.e., to model *all appropriate uses*, though specifications do not ordinarily even try to identify all the relevant circumstantial factors. As well as there being a background set of constraints with respect to which a model is formulated, there is also a background set of assumptions on which a specification is allowed at any point to rely.

The ultimate conclusion is inescapable. The model of a refrigerator as a device that always maintains an internal temperature of between three and six degrees is but the merest inchoate gesture towards what in full glory is probably impossible: a full specification of refrigeratorhood, suitable to serve as the basis for air-tight proofs.

7 The Limits of Correctness

It is time to summarize what we have said so far. The first challenge to developing a perfectly "correct" computer system stems from the sheer complexity of real-world tasks. We mentioned at the outset various factors that contribute to this complexity: human interaction, unpredictable factors of setting, hardware problems, difficulties in identifying salient levels of abstraction, etc. Nor is this complexity of only theoretical concern. A December 1984 report of the American Defense Science Board Task Force on "Military Applications of New-Generation Computing Technologies" identifies the following gap between current laboratory demonstrations and what will be required for successful military applications—applications they call "Real World; Life or Death." In their estimation the mid-1980s military needs (and, so far as one can tell, expects to produce) an increase in the power of computer systems of *nine decimal orders of magnitude*, accounting for both speed and amount of information to be processed. That is a one billion (1,000,000,000) fold increase over current research systems, equivalent to the difference between a full century of the entire New York metropolitan area, compared to one day in the life of a hamlet of one hundred people. And remember that even current systems are already several orders of magnitude more complex that those for which we can currently develop proofs of relative consistency. So, to put the point starkly, expected need outstrips current capability by a factor of approximately a trillion.

But sheer complexity has not been our primary subject matter. The second and more serious challenge to computational correctness comes from the problem of formulating or specifying an appropriate model. And the point we have been making is that, except in the most highly artificial or constrained domains, modelling an embedding situation is inherently an approximate, fraught, and compromised task—not a form or complete and perfectible endeavour.

The situations in which modeling has the best hopes of even partial success are those that Winograd has called "**systematic domains**":[16] areas where the relevant stock of objects, properties, and relationships are most clearly and regularly predefined. Thus bacteria, or warehouse inventories, or even flight paths of airplanes coming into airports, are relatively systematic domains, at

[16] «Ref: probably either in *Bringing Design to Software* or in *Understanding Computers and Cognition : A New Foundation for Design*»

least compared to conflict negotiations, any situations involving intentional human agency, learning and instruction, and so forth. The systems that land airplanes are hybrids—combinations of computers and people—exactly because the unforeseeable happens, because what happens is often the result of human action, and because what it is that has happened often requires human interpretation. Although it is impressive how well the phone companies can model telephone connections, lines, and even develop statistical models of telephone use, at a certain level of abstraction, it would nevertheless be impossible to model the *content* of the telephone conversations themselves—what people actually *say*.

Third, and finally, there is the question of what one does about these first two facts. It is because of the answer to this last question that I have talked, so far, somewhat interchangeably about people and computers. With respect to the ultimate limits of models and conceptualization, *both people and computers are restrained by the same truths*. If the world is infinitely rich and variegated—which I not only believe, but would also argue that experience has demonstrated to be pragmatically (if not metaphysically) evident—then no prior conceptualization of it, nor any abstraction, will *ever* do it full justice. That is OK—or at least we might as well say that it is OK, since that is the world we have got. What matters is that we *never forget about that richness*—that we never think, with misplaced optimism, that machines might magically have access to a kind of "correctness" to which people cannot even aspire.

It is time, to put this another way, that we change the traditional terms of the debate. The question is not whether machines can do things, as if, in the background, lies the implicit assumption that people are not only the object of *comparison*, but that the only choice in front of us is whether an assumed action should be taken by a person or by an automated system. The very idea of building an automated system capable, within a few short seconds, of making a "decision" to annihilate Europe, say, should make you uneasy. Requiring a person to make the same decision, also in a matter of the same few seconds, *should also make you uneasy*—and for very similar reasons.

Fundamentally—to say what is obvious but somehow also extremely worth saying—a decision to annihilate Europe *should never be made within a few short seconds*.[17] It should never be made because there is no way that reasoning of any sort, be it human or machine, could possibly do justice to the inevitable complexity of the situation, because of fundamental metaphysical facts about how reasoning relates to the world. Because reasoning is based on partial models, it is an ultimate and inherent truth that reasoning can never be guaranteed to be correct in the sense of doing full justice to what is the case.

Which means, to suggest just one possible strategy for action, that we might try, in our treaty negotiations, to find mechanisms to slow our weapons systems down.

It is striking to realise, once the comparison between machines and people is raised explicitly, that we do not typically expect "correctness" for people in anything like the form that that we presume it for computers. In fact quite the opposite, and in a revealing way. Imagine, in a by-gone era, sending a soldier off to war, and giving him (it would surely have been a "him," then) his final instructions. "Obey your commander; help your fellow-soldier," you might say, "*and above all do your country honour*". What is striking about the last clause is what it betrays: the recognition that it is considered not just a weakness, but a punishable weakness—a breach of morality—to obey instructions absolutely blindly (in fact, and for relevant reasons, it is generally *impossible* to follow instructions blindly; they have to be interpreted to the situation at hand). Soldiers are subject to court martial, for example, if they violate fundamental moral principles, such as murdering women and children, even if following strict orders.

In the human case, in other words, most our social and moral systems, even including the strict disciplinary institutions of the military, have built in into them an acceptance of the uncertainties and limitations inherent in the model-world relationship (relation b in figure 1). We *know* that the assumptions and preconceptions built into instructions will sometimes fail, and we *know* that instructions are always incomplete. We exactly rely on judgment, responsibility, consciousness, and so forth, to carry some-

[17] If, of course, ever, and at all.

one through those situations where model and world part company—which is to say, through all situations, since model and world always part company, to a lesser or greater extent.

Saliently, in fact, we never talk about people, in terms of their overall personality, being *correct*. It is only concrete individual actions, fully situated in particular settings, that are (or are not) correct —not people in general, or systems. Rather, when people are the subject matter, we speak of their being **reliable**—a much more substantial term. What leads to the highest number of correct human actions is a person's being reliable, experienced, capable of good judgment, etc., so that, as often as possible, and to the greatest extent possible, based on partial, incomplete, and likely fallible information about stupefyingly complex real situations, they can aim as strenuously as possible towards doing the right thing.

There are two possible morals here, for computers. The first has to do with the notion of **experience**. In point of fact, program verification is not the only, or even the most common, method of obtaining assurance that a computer system will do the right thing. In the real world, programs are usually judged acceptable, and are typically accepted into use, not because we prove them "correct," but because they have *shown themselves relatively reliable* in their destined situations, for some substantial period of time. And, as part of this experience, we expect them to fail: there always has to be room for failure. Certainly no one would ever accept a program without such *in situ* testing: a proof of correctness is at best added insurance, not a replacement, for real life experience. Unfortunately, however, for the ten million lines of code that is proposed to control and coordinate the Star Wars Defense System, there will never, God willing, be an *in situ* test.

One answer, of course, if genuine testing is impossible, is to run a *simulation* of the real situation. But even at its best, simulation, as our diagram should make clear, *can also test only the left-hand side relationship*. Simulations are defined in terms of models; that is what a simulation is: *a concretization of a model*. It is not an actual yet somehow not actual real world. As a result, simulations do not and cannot test relationships between models and world. That is exactly why simulations and tests can never replace real-world *in situ* testing—cannot replace embedding a program in the real world and seeing how it behaves. All the war games we hear about, and hypothetical military scenarios, and electronic battlefield simulators, and so forth, are all based on exactly the kinds of models we have been talking about all along. In fact the subject of simulation, worthy of a whole analysis on its own, is really just our whole subject welling up all over again in what is only a superficially different guise.

I said earlier that there were two morals to be drawn, for the computer, from the fact that we ask people to be *reliable*, not to be *correct*. The second moral is for those who, when confronted with the fact that genuine or adequate experience cannot be had, would say "Well if that is true, let's build responsibility and morality into computers. If people can have it, there is no reason why machines cannot have it too."

I will not argue that building responsibility and morality into artefacts is inherently impossible, in some metaphysical or ultimate philosophical sense, but lest anyone be tempted in that direction, a few short comments are in order. First, from the fact that humans sometimes *are* responsible, it does not follow that we know what responsibility is: from tacit skills no explicit model is necessarily forthcoming. We simply do not know what aspects of the human condition underlie the modest levels of responsibility to which we sometimes rise. Second, with respect to the goal of building computers with even human levels of full reliability and responsibility, I can state with surety that the present state of artificial intelligence is about as far from this as mosquitoes are from flying to the moon. Whether it will be 50 or 500 years before we have responsible machines around is a topic we could debate, but no one currently alive need worry about what it will be like to live with them.

But there are deeper morals even than these. The point is that even if we could make computers reliable, they still would not necessarily always do the correct thing. Remember: *people* are not always "correct", either; correctness at that level is not something this world will ever provide. That is why we hope that people are, and educate them to be, responsible. And if civilization has learned anything over the past few millennia, it is surely that correctness and responsibility do not always coincide. Even if, in another thousand years, someone were to devise a genuinely responsible computer system, there is no reason to suppose that it would achieve "perfect correctness" either, in the sense of never doing anything wrong. This is not failure, in the sense of a performance limitation; it stems from the deeper metaphysical fact that models must be abstract, in order to be useful. This is the lesson to be learned from the violence inherent in the model-world relationship: that there is an *inherent* conflict between the power of analysis and conceptualization, on the one hand, and sensitivity to the infinite richness, on the other.

But perhaps this is an overly abstract way to put it. Perhaps, instead, we should just remember that there will always be another moon-rise.

7 — One Hundred Billion Lines of C++[†]

The year is 2073. You have a job working for General Electric, designing fuel cells. Martian have landed. One stands over your desk, demanding to see what you are working on. On the large CAD display surface forming your desk, you are sketching a complex combustion chamber for a new eco-engine you and some colleagues are designing. Next to an input port, on the left side, is the word 'oxygen,' with an arrow pointing inwards. On the right is a similar port, with the word 'hydrogen.' "Amazing!," says the Martian to a conspecific, later that day. "Earthlings build symbol combustion machines! I saw some engineers designing one. They showed me how the word 'oxygen' would be combined with the word 'hydrogen' in a wondrous kind of symbol mixing chamber."

The Martian is confused. That was a *diagram* for a fuel cell, not a fuel cell itself. The word 'oxygen' was a label. Map is not territory. What will be funneled into the input chamber—to belabour the obvious—is oxygen gas, not (a token of) the word 'oxygen.' Words entering chambers makes no sense.

Far-fetched? Perhaps. But in this paper I argue that the debate that has been conducted, over the last decade or so, between symbolists and connectionists founders over a troublingly similar error. Perhaps not quite as egregious—but a misunderstanding, nonetheless. Moreover, the confusion goes far beyond that particular debate, infecting (mis)understandings of the computational theory of mind throughout philosophy—including, to take

[†] Published in *Cog Sci News*, Lehigh University, 1997. Thanks to Irene Appelbaum and Güven Güzeldere for comments on an early draft.

just one example, the debate about Searle's notorious Chinese Room. It is as if John Searle had wandered into a hacker's office, looked over her shoulder at the program she was writing, seen lots of symbols arranged on the screen, and concluded that the resulting system must be symbolic. Searle's inference, I claim, is no more valid than the Martian's.

For discussion, I will focus on the connectionist debate, but the points can easily be extended to other contexts.

1 Background

A glimmer of trouble is evident in the way the connectionist debate is framed. Both positions consider only two kinds of architecture. On one side are traditional von Neumann architectures, of the sort imagined in "good old fashioned ai" ('GOFAI,' to use Haugeland's term). These systems are assumed to be constructed out of a set of atomic symbols, combined in countless ways by rules of composition, in the way that is paradigmatically exemplified by the axioms of a first-order theorem prover. On the other side are connectionist (or dynamic) systems, composed instead of a web of interconnected nodes, each dynamically assigned a numerical weight. For purposes of this debate, it seems as if that is all there is. Some writers[1] even take the first, symbolic, model, to be synonymous with computation *tout court*. So they frame the argument this way: that cognition is (should be understood as, will best succumb to analysis as, etc.) a *dynamical* system, not a *computational* system.

What happens to real-world programming in this scheme—the uncountably many network routers and video games and disk compression schemes and e-mail programs and operating systems and so on and so forth, that are the stock and trade of practicing programmers? Which side of the debate are they on? Most people, I take it, assume that they fall on the symbolic side. But is that so? And if so, why are such systems never mentioned?

It cannot be that they are not mentioned because such programs are rare. In National Public Radio's famous phrase, "let's do the

[1] E.g. Port, Robert and van Gelder, Timothy (eds.), *Mind as Motion*, Cambridge, Mass.: MIT Press (1995), or van Gelder, Timothy "Computation and Dynamics," *Journal of Philosophy*, …

numbers."[2] Sure enough, some combinatorial symbolic systems have been constructed, over the years, of just the sort envisaged (and defended) by Fodor, Pylyshyn, and others on the symbolic side of the debate.[3] Logic-based programs, theorem provers, and knowledge representation systems were early examples. SOAR[4] is a more modern instance, as is the CYC project of Lenat and Feigenbaum. Perhaps the category should even be taken to include the bulk of expert systems, case-based reasoners, truth-maintenance systems, and diagnosis programs. What does this come to, overall? Perhaps somewhere between 1,000 and 10,000 programs? Suppose each comprises an average of 10,000 lines of code (a couple of hundred pages, in normal formatting). That would come to ten million lines of code, overall.

But now consider the bulk of real-world programming. Think of e-mail clients, of network routers, of word processors and spreadsheets and calendar programs, of operating systems and just-in-time compilers, of Java applets and network agents, of embedded programs that run the brakes in our cars, control traffic lights, and hand your cellular telephone call from one zone to the next, invisibly, as you drive down the interstate. Think, that is, of commercial software. Such programs constitute far and away the mainstay of computing. Again, it is impossible to make even much of a rough estimate, but it will not be too misleading if we assume that there are probably something on the order of 10^{11}—i.e., one hundred billion—lines of C++ code in the world.[5] And we are barely started.

In sum: symbolic AI systems constitute approximately 0.01% of

[2] «Ref 'Marketplace'»

[3] See for example Pinker, Steve, and Mehler, Jacques (eds.), *Connections and Symbols*, Cambridge, Mass.: MIT Press, 1988.

[4] '«ref»

[5] It is not even clear how one would individuate programs—or, for that matters, lines of code. When does one line turn into another one? How long does a line have to exist (e.g., in a rough-draft of a program, in a throw-away implementation) in order to count? What about multiple copies? Moreover, since C++ is already passé, what about Java? Or the language that will be invented after that?

I have no clue as to how to answer such questions. Maybe this is a better estimate: $10^{9\pm(3\pm2)}$. Whatever; the answers do not matter to any of the points being made in the text.

written software.

By themselves, the numbers do not matter. What I want to do is to use these facts to support the following claims:

1. Within the overall space of possible computational architectures, the vast majority of commercial software—which is to say, the vast majority of software, period—is neither "symbolic," in the sense defended by Fodor and Pylyshyn, nor "connectionist," in the sense defended by Smolensky, nor "dynamic," in the sense advocated by van Gelder, but rather some fourth kind entirely;
2. The only reason for thinking that commercial software is symbolic, as we will see, stems from a confusion between a **program** and the **process** or computation that it specifies (something of a use/mention error, not unlike that made by the Martian); and
3. In order to understand how such a confusion could be so endemic in the literature (and have remain so unremarked), one needs to understand that the word "semantics" is used differently in computer science from how it is used in logic, philosophy, and cognitive science—a requirement that in turn will require us to understand something about the history of the technical vocabulary used in computer science.

In a sense, the ultimate moral comes to this: the "design space" of possible representational/computational systems is enormous—far larger than non-computer-scientists may realize. Both the traditional "symbolic" variety of system, as imagined in GOFAI, and the currently-popular connectionist and dynamic architectures, are *just two tiny regions*, of almost vanishingly small total extent, within this vast space.

Within the hugely important project of exploring how human cognition works, it may be important, or anyway of moderate interest, to ask whether and how much human cognition fits within these regions—to what extent, in what circumstances, with respect to what sorts of capacities, etc. But to assume that the two represent the entire space, or even a very large fraction of the space—even to assume that they are especially important anchor points in terms of which to dimension the space—is a mistake. Our imaginations need to run much freer than that.

And commercial software shows us the way.

2 Compositionality

What it is that defines the symbolic model is itself a matter of debate. But as Fodor and Pylyshyn make clear, there are several strands to the basic picture:

1. It is assumed that there exist a relatively small (perhaps finite) stock of basic representational ingredients: something like words, atoms, or other entities we can call **simplexes**.
2. There are **grammatical formation rules**, specifying how two or more representational structures can be put together to make **complexes**.[6]
3. It is assumed that the simplexes have some **meaning** or **semantic content**: something in the world that they mean, denote, represent, or signify.
4. Finally—and crucially—the meanings of the complexes are assumed to be built up, in a **systematic way**, from the meanings of the constituents.

The picture is thus somewhat algebraic or molecular: you have a stock of ingredients of various basic types, which can be put together in an almost limitless variety of ways, in order to mean or represent whatever you please. This "compositional" structure[7]

[6] Words of English—or anyway their morphological stems—are good examples of simplexes; and sentences and other complex phrases of natural language are good examples of complexes. But words have various additional properties—such as having spellings, being formulable in a consensual medium between and among people so as to serve as vehicles for communication, etc.—that are not taken to be essential to the symbolic paradigm.

[7] Compositionality is a complex notion, but is typically understood to consist of two aspects: first, a syntactic or structural aspect, consisting of a form of "composition" whereby representational symbols or vehicles are put together in a systematic way (according to what are often known as formation rules), and a semantic aspect, whereby the meaning or interpretation or content of the resulting complex is systematically formed out of

underwrites two properties that Fodor identifies as critical aspects of human thinking: **productivity** (the fact that we can produce and understand an enormous variety of sentences, including examples that have never before occurred) and **systematicity** (the fact that the meaning of large complexes is systematically related to the meanings of their parts). Much the same structure is taken by such writers as Evans and Cussins[8] to underlie what is called *conceptual representation*. The basic idea is that your concepts come in a variety of kinds: some for individual objects, some for properties or types, some for collections, etc.; and that they, too, can similarly be rearranged and composed essentially at will. So a representation with the content $P(x)$ is said to be *conceptual*, for agent A, just in case: for every other object x', x'', etc. that A can represent, A can also represent $P(x')$, $P(x'')$, etc., and for every other property P', P'', etc. that A can represent, A can also represent $P'(x)$, $P''(x)$, etc.[9]

Thus suppose we can say (or entertain the thought) that a table is 29" high, and that a book is stolen. So too, it is claimed—given that thought at this level is conceptual—we can also say (or entertain the thought that) the table is stolen and the book is 29" high (even if the latter does not make a whole lot of sense). This condition, called the "**Generality Condition**" by Evans, is taken to underwrite the productive power of natural language and rational thought. It is also clearly a property taken to hold of the paradigmatic instances of "symbolic" AI—i.e., of logical axiomatisations, knowledge representation systems, and the like. Whether being compositional and productive is considered to be a *feature*, as Fodor suggests, or a *non-feature*, as various defenders of non-conceptual content suggest—i.e., whether it is viewed positively or negatively—there is widespread agreement that it is an important property of some representation schemes, and paradigmatically

the meanings or interpretations or contents of its constituents, in systematic way (in a way, furthermore, associated with the particular formation rule the complex instantiates).

[8] Evans, Gareth, *Varieties of Reference*, Oxford: Clarendon Press (1982); Cussins, Adrian, "On the Connectionist Construction of Concepts," in Boden, Margaret. (ed.), *The Philosophy of Artificial Intelligence*, New York: Oxford University Press (1990).

[9] Evans says 'entertain the judgment' that *a* is *F*, that *b* is *G*, etc., rather than 'represent'; I use the representational phrasing here since the subject matter is symbolic computation.

exemplified by ordinary logic. Indeed, the converse, while too strong, is not far from the truth: some people believe that connectionist, "subsymbolic," "non-symbolic" and other forms of dynamical system are recommended exactly in virtue of being non-compositional or non-conceptual.

3 Programs

What about those billions of lines of C++ code? Are they conceptual, in this compositional sense?

We need a distinction. Sure enough, the programming language C++ is a perfect example of a symbolic system. An indefinite stock of atomic symbols is made available, called *identifiers*, some of which are primitive, others of which can be defined. There are (rather complex) syntactic formation rules, which show how to make complex structures, such as conditionals, assignment statements, procedure definitions, etc., out of simpler ones. Any arrangement of identifiers and keywords that matches the formation rules is considered to be a well-formed C++ program—and will thus, one can presume, be compiled and run. By far the majority of the resulting programs will do nothing of interest, of course—just as by far the majority of syntactically legal arrangements of English words make no sense. But it is important that these possible combinations are all *legal*. That is exactly what makes programming languages so powerful.

But—and this matters—it does not follow that most commercial *software* is symbolic. For consider the language used in that last paragraph. What is compositional—and hence is symbolic—is *the programming language*, taken as a whole, *not any specific program that one writes in that language*. It follows that *the activity of programming* is a symbolic process—i.e., the activity engaged in by people, for which they are often well paid. That may be an important fact, for a variety of reasons: it might be usable as an early indicator of what children will grow up to be good programmers, or represent an insight into or limitation on how we construct computers. But it is irrelevant to the computational theory of mind, since it is not *programming* that mentation is supposed to be like, according cognitivism's fundamental thesis.[10] Rather, the claim of

[10] It is by no means clear that programming is a *computational* activity.

the computational theory of mind is that thought or cognition or mentation is like (or even: is) *the running of a (single) program*.

Thus if you write a network control program, and I write a hyperbolic browser, and a friend writes a just-in-time compiler, all in C++, each of us uses the compositional power of the C++ programming language to specify a particular computational program or process or architecture. There is no reason to suppose—good reason not to suppose, in fact—that those programs, those resulting specific, concrete active loci of behavior, *will retain the compositional power of the language we used to specify them*. To think so is, like the Martian, to make something of a use/mention mistake.

To make this precise, we need to be more careful with our language. As is entirely standard, I will call C++ and its ilk (Fortran, Basic, Java, JavaScript, etc.) **programming languages**. As stated above, I admit that programming languages are compositional representational systems—and hence symbolic. They are used, by people, to specify or construct individual programs. Programs are static, or at least passive, roughly textual, entities, of the sort that you read, edit, print out, etc.—i.e., of the sort that exists in your EMACS buffer.[11]

What programs are for is to produce behavior. That is why we write them. Behavior is derived from programs by *executing* or *running* them. Programs can be executed directly in one of two ways: (i) they can be executed by the underlying hardware of the machine, if they are written in the lowest level language (called 'machine language'), in which case the term 'execution' is the most common one used; or (ii) they can be executed by another computational process, which itself results (directly or indirectly) form the execution of a machine language program, in which case the execution of the (higher-level) program is typically called **inter-**

Chances are, programming will turn out to be to be computational if and only if cognitivism is true.

[11] Technically, a distinction needs to be made between the program at the level of abstraction (and internal implementation) that a compiler can see—the one that gets "written" on a computer's hard disk, etc.—and the strictly "print representation" in ASCII letters, that people can read. For purposes of this paper, however, this distinction, too, does not matter. As is common parlance, therefore, I will refer to both, interchangeable, as "the program."

pretation, and the process that does the execution, the **interpreter**.[12] Of the two, the notion of interpretation is more general; and since most machines, these days, are micro-coded, even (so-called' machine language programs are typically interpreted, but a process resulting from a still-further lower level program, written in what is called 'microcode,' which in turn is directly executed by the microcode hardware.

Commonly, however, programs are not directly executed. Instead, they are first *translated*, by a process called **compilation**, into another language more appropriate for direct execution by a machine. That is, if program P_1 is written in C++, instead of being run or executed directly, by a C++ interpreter, it will instead be translated into another program P_2, perhaps in machine language, such that the execution of P_2 results in the "same" behaviour as would have resulted by the direct execution of P_1 by a C++ interpreter.

However it comes into existence, the ultimately resulting behavior—the whole point of the exercise—is what I will call a **process**. When (in the computer scientist's sense of that term) a program is *interpreted*, therefore, to put this all simply, what results is *behavior* or a *process*. But when a program is *compiled*, what results is not behavior, but another program, in a different language (typically: machine language). When that machine language program is executed, however, once again a process (or behavior) will result.

For our purposes, having to do with what is and is not symbolic, what matters is that once a program is created, *its structure is fixed*. Except in esoteric cases of reflective and self-modifying behavior—which is to say, except in a vanishingly small fraction of those 10^{11} lines of code—the entire productive, systematic, compositional power of the programming language is set aside when the program is complete. The process that results from running that program is...well, whatever the program specifies. But, at least to a first order of approximation, the compositional power of the programming language is as irrelevant to the resulting process as the compositional and productive power of a computer-aided design system (CAD) is irrelevant to the thereby-specified fuel cell.

[12] Why this is called *interpretation* will be discussed in the next section.

Consider an example. Suppose we are writing a driver for a print server, and need to represent the information as to whether the printer we are currently servicing is powered up. It would be ordinary programming practice to define a variable called current-printer11 to represent whatever printer is currently being serviced, and a predicate called PoweredUp? to be the Boolean test. This would support the following sort of code:[13]

```
if PoweredUp?(current-printer)
        then … print out the file …
        else TellUser ("Printer not powered on. Sorry.")
```

But now consider what happens when this program is compiled. Since the question of whether or not a printer is powered up is a Boolean matter, the compiler is free to allocate *a single bit* in the machine (per printer) to represent it. That will work so long as the hardware is arranged to ensure that whenever the printer is powered up, the bit is set (say) to '1'; otherwise, it should be set to '0'. Instances of calls to `PoweredUp?` can then be translated into simple and direct accesses of that single bit. In the code fragment above, for example, if that bit is 1, the file will be printed; if it is a 0, the user will be given an error message. And so all the compiler needs to produce is a machine whose behavior is functionally dependent on the state of that bit in some way or other.

This is all straightforward—even elementary. But think of its significance. In particular Consider Evans' Generality Condition, described above. In order for a system to be compositional in the requisite way, what was required, was the following: that the system be able to "entertain" a thought—construct a representation, say—whose content is that any property it knows about hold of any object it knows about. Suppose, for argument, that we say that the print driver "knows about" the current printer, and also "knows about" the user—the person who has requested the print job, to whom the potential error message will be directed. Suppose, further, that we say that the driver, as written, can "entertain the thought" that the printer is powered up. Does that imply that it can entertain a thought (or construct a representation) whose content is that the user is powered up?

Of course not. In fact the print driver process cannot entertain a

[13] By design, this code fragment is ridiculously skeletal.

single "thought" that does not occur in the program. That shows that it is not really "entertaining" the thought at all. For the issue of whether the printer is powered up is not a proposition that can figure, arbitrarily, in the print driver's deliberations. In a sense, the print driver doesn't "deliberate" at all. It is a machine, designed for a single purpose. And that is why the representation of whether a given printer is powered up can be reduced to a single bit. It can be reduced to a single bit because the program has absolutely no flexibility in using it. Sure, given that C++is incontestably symbolic, productive, and so forth, the original programmer could have written any of an unlimited set of other programs, rather than the program they wrote. But *given that they wrote the particular one that they did*, that extrinsic flexibility is essentially irrelevant.

From one point if view, in fact, that is exactly why we compile programs: to get rid of the overhead that is required in the original programming language to keep open (for the programmer) the vast combinatoric space of possible programs. Once a particular program is written, this space of other possibilities is no longer of interest. In fact it is in the way. It is part of the compiler's task to wash away as many traces of that original flexibility as possible, in order to produce a sleeker, more efficient machine.

Another numerical point will help drive the point home. Programs to control high-end networked printers are several million lines long. Operating systems are 100s of millions of lines of code.[14] It is not unreasonable to suppose that such programs contain a new identifier every four or five lines. That suggests that the number of identifiers used in a printer control program can approach a million, and that Windows NT will contain as many as 7 million identifiers. Suppose a person's conceptual repertoire is approximately the same size as their linguistic vocabulary. Educated people typically know something like 40,000 to 80,000 words. Suppose we therefore assume that people have on the order of 100,000 concepts. Is it possible—as seems to be entailed by the symbolists' position—that a Xerox printer has a conceptual reper-

[14] Microsoft Windows NT 5.0, the release of which was thought to be imminent at the point when this paper was first written, was rumoured to contain 35 million lines of code. (It was eventually released on February 17, 2000.)

toire ten times larger than you do, or a Microsoft operating system, seventy times larger?

I think not.[15]

4 Processes

A way to understand what is going on is given in figure 1. The box at the top left is (a label for!) the program: the passive textual entity selected out of the vast space of possible programs implicitly provided by the background programming language. The cloud at the middle right is intended to signify the process or behavior that results from running the program.[16] The scene at the bottom is a picture of the program's task domain or subject matter. For example in this case the process might be an architectural system dealing with house design.

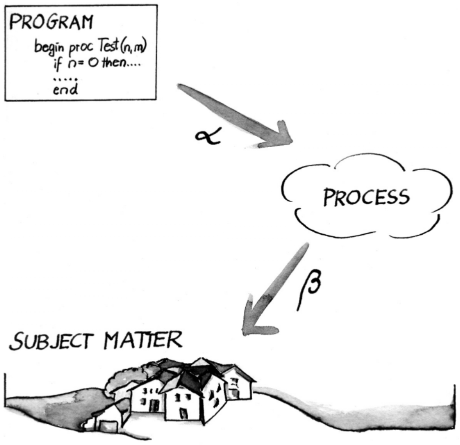


Figure 1 — Program and Process

Given these three entities, two relations are most important: that labelled a, from program to process, and that labelled b, from resulting process to task domain. Moreover, what is perhaps the single most confusing fact in cognitive science's use of computation is this: *the word 'semantics' is used by different people for both of these relations.* In

[15] Indeed, no program—at least none we currently know how to build—could possibly cope with millions of differently-signifying identifiers, if all those identifiers could be mixed and matched, in a compositional way, as envisaged in the symbolists' imagination.

[16] Whether the cloud represents a single run (execution) of the process, or a more general abstract type, of which individual runs are instances, is an orthogonal issue—important in general, but immaterial to the current argument.

computer science, the phrase "the semantics of a program" refers to the program-behavior (process) relation a, whereas the relation considered semantic in the philosophy of mind is the process-world relation b. For discussion, in order not to confuse them, I will refer to a as **program semantics**, and to b as **process semantics**. It is essential to realize that they are not the same.[17] Not only do they relate different things, but they are subject to vastly different constraints—and are of distinct metaphysical kinds.

All sorts of confusion can be cleared up with just this one distinction. But a cautionary note is needed first. Given that processes and behaviours are computer science's primary subject matter, you might think that there would be a standard way to describe them. Curiously enough, however, that is not so. Rather, professional practice instead *models* processes in various ways:

… In the final version it will probably be helpful to devote more than a sentence to each of these; perhaps even worth constructing a target program P that does something (a bit more complex than the printer example above), and then actually presenting the five different models of the processes that result. …

1. The most common way to talk about processes is to model them with (mathematical) functions mapping their inputs onto their outputs.
2. A second way is to treat the computer as a state machine, and then to view the process or behaviour as a sequence of state changes.
3. A third is to have the process produce a linear record of everything that it does (called a "dribble" or "log" file), and to model the process in its terms.

[17] Many years ago, at Stanford's Center for the Study of Language and Information (CLSI), I, with a background in AI and philosophy of mind, tried in vain to communicate about semantics with Gordon Plotkin, one of the most preëminent theoretical semanticists in all of computer science. Finally, a glimmer of genuine communication transpired when I came to understand the picture sketched in figure 1, and realised that we were using the term 'semantics' differently. "What I am studying," I said, trying to put it in his language, "is the semantics of the semantics of programs."

Plotkin smiled.

4. A fourth (called "operational semantics") is to model the process in terms of a different program in a different language that would, if run, generate the same behavior as the original.

5. A fifth and particularly important one—called **denotational semantics**—models the concrete activity that the program actually produces (i.e., the behaviour Q) with various abstract mathematical structures (such as lattices), rather in the way that physicists model concrete reality with similarly abstract mathematical structures (tensors, vector fields, etc.).

Especially because of the common use of mathematical models in several of these approaches (#s 1 and 5 especially, though they can all be mathematized), outsiders are sometimes tempted to think that computer science's notion of semantics is similar or equivalent to that used in logic and model theory. But that assumption is misleading. Although the relation is studied in a familiar way, what relation it is that is so studied may differ substantially from what is supposed.

5 Discussion

Once these modelling issues are sorted out, we can use these basic distinctions they are defined in terms of to make the following points:

… This section has not really been written; the six points identified below should be amplified enough to communicate the essential moral, in each case, to someone who does not "already know it," as it were—in particular, enough detail both to motivate and to convey it to a philosophical reader, even one without computational experience …

1. (Discussed above) It is *programs*, not *processes*, that, in standard computational practice, are symbolic (compositional, productive, etc.).

2. It is again *programs*, not *processes*, that computer scientists take to be syntactic. It strikes the ear of a computer scientist oddly to say that a process or behavior is syntactic. But

when Fodor talks about the language of thought, and argues that thinking is formal, what he means, of course, is that human thought processes are syntactic.

3. Searle's analogy of the mind to a program is misleading.[18] What is analogous to mind, if anything (i.e., if the computational theory of mind is true) is *process*.

4. Not only is there no reason to suppose, but in fact I know of no one who ever *has* proposed, that there should be a *program* for the human mind, in the sense we are using here: a syntactic, static entity, which specifies, out of a vast combinatoric realm of possibilities provided for by the programming language, the one particular architecture that the mind in fact instantiates. Perhaps cognitive scientists will ultimately devise such a program. But it seems relatively unimaginable that evolution constructed us by writing one.[19]

5. For simple engineering reasons, the program-process relation (a in the figure) must be constrained to being *effective* (how else would the program run?). There is no reason to suppose that the process-world relation b need be effective, however—unless for some reason one were metaphysically committed to such a world view.

6. It is because computational semanticists study the program-process relation a, not the process-world relation b, that theoretical computer science makes such heavy use of intuitionistic logic (type theory, Girard's linear logic, etc.) and constructive mathematics.

6 Conclusion

… Once §5 is properly written, this § will deserve a rewrite …

[18] Searle, John, *Minds, Brains, and Science*, Cambridge: Harvard University Press (1984).

[19] Of course one could call DNA a programming language in this sense…«talk about how it is subject to some of the same efficacy constraints»

What, in sum, can we say about the cognitive case? Two things, one negative, one positive. On the negative side, it must be recognized that it is a mistake to assume that modern commercial programming gives rise to processes that satisfy anything like the defining characteristics of the "symbolic" paradigm. Perhaps someone could argue that most—even all—of present-day computational processes are symbolic on some much more generalized notion of symbol.[20] But the more focused moral remains: the vast majority of extant computer systems are not symbolic in the sense of "symbol" that figures in the "symbolic vs. connectionist" or "computational vs. dynamic" debates.

What are the computer systems we use, then? Are they connectionist? No, of course not. Rather—this is the positive moral—they spread out across a extraordinarily wide space of possibilities. With respect to the full range of computational possibility, moreover, present practice may not amount to much. Computation is still in its infancy; we have presumably explored only a tiny subset of the space—perhaps not even a very theoretically interesting subset, at that. But this much we can know, already; the space that has already been explored is far wider than debates in the cognitive sciences have so far recognized.

[20] If it were enough, in order to be a symbol, to be discrete and to carry information, then (at least arguably) most modern computational processes would count as symbolic. Or at least that would be true if computation were discrete—another myth, I believe (see chapter ■■). But the symbolic vs. connectionist and/or dynamicist debate is not simply a debate about discrete vs. continuous systems.

8 — The Semantics of Clocks[†]

> *The inexorable ticking of the clock may have had more to do with the weakening of God's supremacy than all the treatises produced by the philosophers of the Enlightenment . . . Perhaps Moses should have included another Commandment: Thou shalt not make mechanical representations of time.*
>
> —Neil Postman[1]

1 Introduction

Clocks?

Yes, because they participate in their subject matter, and participation—at least so I will argue—is an important semantical phenomenon.

To start with, clocks are about time; they represent it.[2] Not only that, clocks themselves are temporal, as anyone knows who, wondering whether a watch is still working, has paused for a second or two, to see whether the second hand moves. In some sense everything is temporal, from the price of gold to the most passive rock, manifesting such properties as fluctuating wildly or being

[†] Slightly revised version of a paper that appeared in James H. Fetzer (ed.), *Aspects of Artificial Intelligence*, Kluwer 1998, pp. 3–31.

[1] Postman (1985), pp.11–12.

[2] Clocks represent time for us, as it happens, not for themselves—but that will count as representation, at least here. I am sympathetic to such distinctions as between original and derivative semantics, and between authentic and derived; in fact I am interested in participation in part for just such reasons. However I am against relativizing representation to an observer at the outset, especially to a human observer (cf. Winograd and Flores, 1986), since to do that would be to abandon any hope of explaining how the human mind might itself be representational. See (Smith, forthcoming).

inert. But the temporal nature of clocks is essential to their semantic interpretation, more than for other representations of time, such as calendars. The point is just the obvious one. As time goes by, we require a certain strict coordination The time that a clock represents, at any given moment, is supposed to be the time that it is, at that moment. A clock should indicate 12 o'clock just in case it *is* 12 o'clock.

But that is not all. The time that a clock represents, at a given moment, is also a function of that moment, the very moment it is meant to represent. I.e., suppose that a clock does indicate 12 o'clock at noon. The time that it indicates a moment later will differ by an amount that is not only proportional to, but also dependent on, the intervening passage of time. It does not take God or angels to keep the clock coordinated; so long as the mechanism is set up properly, it does it on its own. This is where participation takes hold.

2010 Perspective[α1]

………… to be written …………

Things to be talked about:

… How the paper emerged in part out of a desire to combine derivatives and semantic brackets, as part of the unification project (based on another bar conversation about using the states of dynamical systems as "representationally significant" syntactic states).

… The (huge) long-term importance of the comment about "state-change,"

… How, in teaching, I use clocks as a first example of a mechanical system designed to honour a non-effective semantical norm (rather than logic, because people get so confused by the notation—as well as the intrusion of mathematics).

… Go over the notes from the phil-comp course where I talk about clocks, and put into these perspective comments anything there that is not covered in the paper (Jun's feeling is that those notes are easier and more important, for students).

Notes

[α1] Sidebars and footnotes with text in sans-serif font, as in this case, contain comments and reflections added in 2010, rather than material that appeared in the original paper.

As well as representing the current time, clocks have to identify its "location" in the complex but familiar cycle of hours, minutes, etc. They have to measure it, that is, in terms of a predetermined set of temporal units, and they measure it by participating in it. And yet the connection between their participation and their content is not absolute—clocks, after all, can be wrong. How it is that clocks can participate and still be wrong is something we will have to explain.

For clocks, participation involves being dynamic: constantly changing state, in virtue of internal temporal properties, in order to maintain the right semantic stance. This dynamic aspect is a substantial, additional, constraint. A passive disk inscribed with 'NOW!' would have both temporal properties mentioned above (being about time, and having the time of interpretation relevant to content) and would even maintain perfect coordination. A rendering of this word in blinking lights, mounted on a chrome pedestal, might even deserves a place on California's Venice Boardwalk. But even though it would be the first time piece in history to be absolutely accurate, such a contraption would not count as a genuine chronometer.

We humans participate in the subject matter of our thoughts, too, when we think about where to look for our glasses, notice that we are repeating ourselves, or pause to ask why a conversant is reacting strangely. Why? What is this participation? It is hard to say exactly, especially because we cannot get outside it, but a sidelong glance suggests a thick and constant interaction between the contents of our thoughts, on the one hand, and both prior and subsequent non-representational activity, on the other, such as walking around, shutting up, or pouring a drink.

Take the glasses example. Suppose, after first noticing their absence, I get up and look on my dresser, asking myself "Are they here?" My asking the question will be a consequence of my wonder, but so will my (non-representational) standing in front of the dresser. Furthermore, the two are related; the word 'here' will depend for its interpretation on where I am standing. And who knows, to drive the example backwards in time, what caused the initial wonder—eye strain, perhaps, or maybe an explicit comment. The point is that the representational and non representational states of participatory systems are inexorably intertwined—they even rest on the same physical substrate. We can put it even more strongly: the physical states that realise our thoughts are caused by non-representational conditions, and engender non-representational consequences, in ways that must be coordinated with the contents of the very representational states they realise. Participation is something like that.

Artificial intelligence (AI) and general computational systems also participate—more and more, in fact, as they emerge from the laboratory and take up residence with us in life itself: landing airplanes, teaching children, launching nuclear weapons. Far from being abstract, computers are part of the world, use energy, affect the social fabric. This participation makes them quite a lot like us, quite unlike the abstract mathematical expression types on which more familiar semantical techniques have been developed.

My real reason for studying clocks, therefore, can be spelled out as follows. First, issues of semantics, and of the relationship between semantics and mechanism, are crucial for AI and cognitive science (this much I take for granted). Second, it is terrifically important to recognise that computational systems participate in the world along with us. That is why they are useful. Third, as I hope this paper will show, participation has major consequences for semantical analysis: it forces us to develop new notions and new vocabulary in terms of which to understand interpretation and behaviour. Clocks are an extremely simple case, with very modest participation. Nonetheless, their simplicity makes them a good foil in terms of which to start the new development

So they are really not such an unlikely subject matter, after all.

2 Inference and Time-keeping

Let's start by reviewing the current state of the semantical art. Consider a familiar, paradigmatic case: a theorem-prover built according to the dictates of traditional mathematical logic. As suggested in figure 1, two relatively independent aspects will be coordinated in such a system First, there is activity or behaviour—what the system does—indicated as ψ (for psychology). All systems, from car engines to biological mechanisms of photosynthesis, of course do something; what distinguishes theorem provers is the fact that their ψ implements (some subset of) the proof-theoretic inference relation (⊢). Second, there is the denotation or interpretation relation, indicated φ (for philosophy), which maps sentences or formulae onto model-theoretic structures of some sort, in terms of which the truth-values of the formulae are determined. In a computer system designed to prove theorems in abstract algebra, for example, the interpretation function would map states of the machine (or states of its language) onto groups, rings, or numbers—the subject matter of the algebraic axioms.

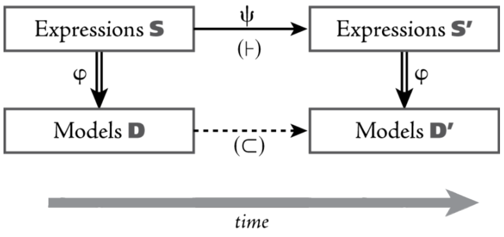


Figure 1 — Activity and semantics for a theorem prover

Four things about this situation are important.

First, although proof theory's putative formality suggests that ψ must be definable independent of φ you could not claim to *have* a proof-theoretic or inference relation except with reference to some underlying notion of semantic interpretation. Conceptually, at the very least, ψ and φ are inextricably linked (salesmen for inference systems without semantics should be reported to the Better Business Bureau). Furthermore, the two relations are coordinated in the well-known way, using notions of soundness and completeness: inferences (ψ) should lead from one set of sentences to another only if the latter are true just in case the former are true (⊢ should honour ⊨). And truth, as we've already said , is defined as in terms of φ: the semantic relation to the world.

Second, even though the proof-theoretic derivability relation (⊢) can be modeled as an abstract set-theoretic relation among sentences, I will view inference itself (ψ) as fundamentally temporal—as an activity. 'Inference' is a derived noun; 'infer' is first and foremost a verb, with an inherent asymmetry corresponding directly to the asymmetry of time itself. It might be possible to realise the provability relation non-temporally, for example by writing consequences of sentences down on a page, but you could hardly claim that the resulting piece of paper was doing inference.

Third, when its dynamic nature is recognised, inference is (quite properly) viewed as a temporal relation between sentences or states of the machine's memory, not as a function from times onto those corresponding sentences or states. Mathematically this may not seem like much of a difference, but conceptually it matters a lot. Thus, taking σ to range over interpretable states of the system, and t over times, ψ is of type $\sigma \rightarrow \sigma$, not $t \rightarrow \sigma$. Of course it will be possible to define a temporal state function of the latter type, which I will call Σ; the point is that it is ψ, not Σ, that warrants the name *inference*. Details will come later, but the relation between the two is roughly as follows: if t' is one temporal unit past t, and $\Sigma(t) = \sigma$, then $\Sigma(t') = \psi(\sigma)$. Inference, that is, has more to do with *changes* in state than with states themselves. To study inference is to study the dynamics of representational systems.

Fourth, of all of the relations in figure 1, only ψ need be effective. Neither φ nor Σ *can be directly implemented or realised,* in the strong sense that there cannot be a procedure that uses these functions' inputs as a way of producing their outputs (the real reason to distinguish ψ and Σ) This claim is obviously true for φ. If I use the name 'Beantown' to refer to Boston, then the relation between my utterance and the town itself is established by all sorts of conventional and structural facts about me, about English, about the situation of my utterance, and so forth. The town itself, however, is not the output of any mechanisable procedure realised in me, in you, or in anyone else (fortunately—as it would be awfully heavy). It might require inference to understand my utterance, but that would only put you in some state σ the same referent as my utterance, or state. In particular, you do not "compute" the referent of an utterance you hear, in the sense of producing that referent as an output of a procedure.[x] Nor is the ref-

[x] While this point is philosophically obvious to the point of banality, there is substantial ambiguity about the word compute. In English we have no difficulty is distinguishing, for example, between 'utter' and 'describe,' in the sense (as I have said elsewhere) one *describes* a refrigerator, and in that act *utters* a sentence (but does not *utter* a refrigerator). Perhaps because so much of computational theory has been developed to deal with mathematical examples, and also because so much computational practice has to do not only with the construction but the representation of computation-internal structures (programs, data structures, etc.), there is no such clarity

erence relation *effectively mediated* by the physical substrate of the world, at least on any understanding of 'effective' remotely connected to the idea that computation has to do with the capcities of *effective* mechanisms. Not even the National Security Agency could fabricate a sensor, to be deployed on route 128, that could detect Boston's participation as a referent in a reference act.[3]

regarding the word 'compute.' People are happy to talk about *computing numbers*, rather than numerals—suggesting it be interpreted as analogous to 'describe'—but also about "computing the header" of a file, in which case it is assumed that the header is actually produced, rather than merely being represented.

[3] In computer science the claim that reference is not computed is viewed suspiciously—for an very interesting reason. To see it, consider why the claim is true. Suppose in a room of one hundred people we label as A the person among them who is the average height. Then suppose a new (101st) person enters the room. Suddenly—*and without any computation*—a different person B will have become the person of the average height. No work needs to be done to "lift" the property of being the average height off of person A, and settling it on B; no energy need be expended; no symbols massaged. The new state just *comes to be*, automatically, in virtue of the maze of conditions and constraints that hold. Reference, I take it, is something like that; conditions and constraints hold so that, when a word is uttered or a thought entertained, some object "becomes" the referent. (Nor is it possible to reply, in the average-height-person case, "Well, the room computed it". On that recourse everything that happens would be computed, which would evacuate the word 'compute' of substance.)

How could computer scientists object to this? For the following reason. Note that the way that B becomes the person of average height is by participating in the situation at hand: he or she *enters the room*. Participation, in other words, is what enables relationship to exist. Computers, on the other hand, are traditionally viewed in purely abstract terms—and abstractions, whatever they are, and whatever else may be true of them, are presumably *metaphysically banned from participation*. The closest an abstraction comes to the property of average height—or indeed to anything at all—is by designating it. And so, because of this abstract conception of computers, one gets lulled into thinking that everything must to come into being in this disconnected, putatively "computational" way.

Needless to say, I do not believe the abstract conception of computers is right. More strongly, I want to argue that participation—virtually the opposite of abstraction—is exactly what allows you to connect to the world in other ways than through explicit symbol manipulation. See section 8, and (Smith, forthcoming).

That Σ is not computed is equally obvious, once you see what it means. The point is a strong metaphysical one: times themselves—metaphysical moments, slices through the *flux quo*—are not causally efficacious constituents of activity; *per se*, they lack causal powers. If they were causally efficacious, clocks would not have been so hard to develop.[4] As it is, mechanisms, like all physical entities, manifest whatever temporal behaviour they do in virtue of momentum, forces acting on them, energy expended, etc., all of which operate *in* time, but do not convert time, compare it to anything else, or react with it. The only thing that is available, as a determiner of how a system is going to be, is how it was a moment before, plus any forces impinging on it (this is physic's vaunted locality). That, fundamentally, is why inference is of type $\sigma \rightarrow \sigma$, not $t \rightarrow \sigma$. *It could not be otherwise.* The inertness of gold, and the indifference of a neutrino, are nothing as compared with the imperturbability of a passing moment

Given these properties of theorem provers, what can we say about clocks? Well, to start with, their situation certainly resembles that of figure 1. As in the inference case, a clock's being in some state σ represents (φ) it's being noon, or 7:15, or whatever; the interpretation function is what matters. Similarly, clocks, like theorem provers, change state (ψ) in a simple but important way. Not only that, state change is what the clock designer has to work with; no mortal machinist. unfortunately, could build a device that would directly implement Σ. Furthermore, as in the case of the theorem prover, the change in state of the clock face is important *only because of its relation to its content.* Forget the Better Business Bureau; no one would *buy* a clock without a clue as to how its state represented time.(without, that is, understanding how it was a *clock*). Once again, systematic coordination between activity and interpretation is what matters.

But despite these similarities, there is a difference between clocks and theorem provers—suggested by the fact that many people (including me)would be reluctant to say that a clock was

[4] For accurately measuring distances on roads, one attaches a "fifth wheel" to a car and reads off the passing miles. Maybe, if time had been causally efficacious, we could have built clocks the same way, running a wheel against time and reading off the passing seconds.

doing *inference*. To get at the difference, note that I have not yet said what inference's coordinated pattern of events is *for* (on the face of it, transitioning from truths to truths sounds a little boring). But the answer is not hard to find: given a set of sentences or axioms that stand in (or enable you to stand in) a given semantical or informational relation to a subject matter, proofs or inference lead you to a *new informational relation* to the same, unchanged subject matter. For example, the famous puzzle of Mr. S and Mr. P[5] focuses your attention on a pair of numbers under a peculiar description; a considerable amount of inference is required in order to give you semantical access *to those same numbers* under a more traditional description (or give you access to other more familiar properties of numbers—there are many ways to discharge the ontological facts). The numbers themselves, however, and their possession of all the relevant properties, are expected to stay put during the inferential process. None of this implies, of course, that the subject matter of inference cannot itself be *temporal*, as illustrated by the situation calculus, temporal logics, and numerous other formal systems. The point is only that the temporality of the inference process and the temporality of the subject domain are not expected to interact.

The situation for clocks, on the other hand, is almost exactly the opposite. What changes, across the time slice mediated by ψ, is not the stance or attitude or property structure that clocks get

[5] There are two numbers between 1 and 100. Mr. P knows their product; Mr. S, their sum. The following conversation ensues:

Mr. P: *I don't know the numbers.*
Mr. S: *I knew you didn't. Neither do I.*
Mr. P: *Now I do*
Mr. S: *Now I do too.*

What are the numbers?

The earliest publication of this problem I am aware of is by H. Freudenthal in the Dutch periodical *Nieuw Archief Voor Wiskunde,* series 3, 17, 1969, p. 152 (a solution by J. Boersma appears in the same series, 18, 1970, pp. 102–106). It was subsequently submitted by David J. Sprows to *Mathematics Magazine* 49(2), March 1976, p. 96 (solution in 50(5) Nov. 1977, p. 268). Perhaps the most widely read version appears in Martin Gardner's "Mathematical Games" in *Scientific American* 241(6), Dec 1979, pp. 22–30, with subsequent discussions and slight variations in 1980: 242(3), March, p. 38; 242(5), May, pp. 24-28; and 242(6), June, p. 32.

at. What changes, rather, is *the subject matter itself*. Clocks never have a moment's rest; no sooner have they achieved the desired relationship to the current time than time slips out from under their fingers—as if God were constantly saying "It's later than you think!" Clocks should perhaps be viewed as the world's first truth maintenance systems: they do what they do merely in order to retain the validity of their single semantic claim. Like any other meter or measuring instrument, they must track the world.

We can summarise:

> *At least as traditionally construed,* **inference** *is a technique that enables a system to* change *its relation to a* fixed *subject matter.* **Clocks** *are almost exact duals: they maintain a* fixed *relation to a* changing *subject matter.*

If reconstructing time-pieces were really my subject matter, rather than simply being a foil, I might stop here. But my real interest is in developing a single semantical framework so that we can not only handle both of these cases (mathematical inference and real-time clocks), but also locate everything in between. So let's spend a minute to see how clocks fit into the general case.

3 Semantically Coherent Activity

I will use the term '**representational system**' to coyer anything whose behaviour fits within the broad space of semantically constrained activity. To be a representational system, in other words, is to be an element of the natural order that acts in a semantically coherent way. Of all possible kinds of representational activity, inference will be analysed as a particular type. The representational space is large, of course, and certainly includes all of computation (more about that in a moment), but it is still a substantive notion: not everything is in it. Planets, for example, are excluded,[x] because planets do not represent their orbits; they just *have* them. Clocks, on the other hand, *do* represent the time, just as I can represent to myself how the sunrise looked this morning, as I drove down from the mountains.

Clocks do however fall outside most traditional models of

[x] Unless accorded semantical significance, which is not usual practice (except perhaps of astrologists).

computation, including the "formal symbol manipulation" model so familiar in cognitive science.[6] First, clocks (their faces, and the clockworks that run them) are fully concrete, physical objects, part of the natural order; nothing abstract here. Furthermore, this concreteness is crucial to our understanding of them; for some purposes one might treat clocks at a level of description that abstracted away from their physical being, including their temporal being, but since our purpose is to show how participation in their subject matter influences their design, to do so would be to miss what matters most. Second, at least some clocks (especially electrical ones operating on alternating current) are analog, even though more and more recent on are "digital."[x] Third, to the extent that clocks have representational ingredients, there is no obvious decoupling to be made between (i) a set of structures that represent, and (ii) an independent process that inspects and manipulates them according to the shapes it sees. In other words, whereas Fodor's characterisation of a computer's "standing in relation" to representational ingredients suggests a modular division between symbols and processor, no such division is to be found in the chronological case. Fourth, there is another separation that cannot be maintained in the case of clocks: that between "internal" and "external" properties. Rather like neutrinos, times permeates everything equally—being as much an influence on internal workings as it is on surrounding context. And of course it is one and the same time, inside and out—clock design depends on this. Fifth, clocks, especially analog clocks, are not usually "programmed" in any sense; they are designed, but they are not universal computers specialised by physical encodings of timekeeping instructions. Like so many other properties of clocks, this is important, and leads to the sixth salient difference. Even on the view that Turing machines are concrete, physical objects (of

[6] The two other primary models, conceptually distinct from the formal symbol manipulation idea, are the automata-theoretic notion of a digital or discrete system, and the related idea of a machine whose behaviour is equivalent to that of some Turing machine. Although the formal symbol manipulation view seems to go virtually unchallenged in cognitive science, the other two have much more currency in modern computer science. See «Smit, forthcoming».

[x] Whatever that means. See «…».

which abstract mathematical quadruples are merely set-theoretic models), there is still no guarantee, given a particular universal one, that *any set of instructions could make it be, or even simulate, an accurate time keeper*—because there need be no consistency or regularity as to how long its state changes take. Turing machines, *qua* Turing machines, do not really participate.

I have come to believe, however, that not one of these properties is essential to the notion of computation on which the economy of Silicon Valley is based, or to the notion that underlies AI's hunch that the mind is computational: (i) being abstract, (ii) being digital, (iii) exhibiting a process/structure dichotomy, (iv) having a clear boundary between inside and outside, (v) being programmable, or (vi) being necessarily equivalent to any Turing machine. Quite the contrary. In (Smith, forthcoming)[x] I argue for a much stronger conclusion: that the only regularity essential to computation has to do with computation's being a *physically embodied representational process*—an active system or process whose behaviour represents some part or aspect of the embedding world in which it participates. Needless to say, this has the consequences of defining computation squarely in terms of undischarged semantical predicates. My position on theoretical cartography is therefore the inverse of Newell's (1980): whereas he thinks that computer science has *answered* the question of what it is to be a symbol, I believe in contrast that the integrity of computation as a notion rests full-square on semantics: it requires a notion of symbol in order to have any foundation. So we have lots of homework to do, but it is homework for another day.

In the meantime, clocks are a good test case for comprehensive semantical frameworks. They lack many important properties of more general computers: they do not act, for example, or have sensors. But since every semantical property they do exhibit is one that computers exhibit too—including participation—they are a useful design study.

[x] Though this paper was written in 1997, this was a reference to AOS «ref».

4 Three Points on Two Factors

In the previous section I distinguished two aspects or factors of any representational system: its behaviour, activity, or causal connection with the world (which I will call the **first factor**) and its interpretation, content, or relation to its subject matter (the **second factor**). I have previously used this two-factor framework to reconstruct the semantics of Lisp, the programming *lingua franca* of AI, and argued for its general utility in analysing knowledge representation systems.[7] And I will use it here, to analyse clocks. But three points must be made clear.

First, the ordering of the two factors may seem odd. There is no doubt that having interpretation or content—standing in semantic relation to a subject matter—is what particularly distinguishes the systems we are interested in. Given this pride of place, it might seem that content should be called first. But for present purposes this would be a mistake. We theoreticians typically treat semantics as primary when we analysing both natural and artifactual languages (such as the predicate calculus). We typically define semantics over rather abstract entities—sentence *types*, for example—and then understandably define the other dimensions (proof theory, inference) over the same domain. But especially in conjunction with the formal-symbol manipulation view of computation, this overall strategy lends a very abstract feel to inference—leading such people as Searle to wonder how, or even whether, such a system could ever possess genume semantical powers. In contrast, by calling activity the first factor here I want to recognise that computational systems are first and foremost, *systems in the world*. Everything has what I am calling a first factor; that is what gives a system the ability to participate. The second factor—of representation or content—which enables a system (a thinker, a clock) to stand in relation to what is not immediately accessible or discriminable, is a subsequent, more sophisticated capacity. It is the second factor, furthermore, that distinguishes the representational or interpretable systems from other natural systems, but it distinguishes them as a sub-type, not as a

[7] Smith (1982, 1984, 1986).

distinct class. First factor participation in the world ("being there," roughly) is always available—which is fortunate, since it is only with respect to the first factor that second factor content can ever be grounded.

In sum, recognising the metaphysical primacy of the first factor is an important ingredient in the defense of naturalism.

Second, there is a natural (almost algebraic) tendency to think that, in accepting a two-factor stance, one is committed to thinking that the two factors, in any given system, will in some important sense be *independent*. This tendency is amplified by the fact that in standard first-order logic an almost total independence of factors is achieved—this is one of the many meanings of the ambiguous claim that first-order logic is **formal**. Truth, content, and interpretation in logic are thought to be relatively independent of proof-theoretic role, and provability or inferential manipulation analogously independent of content or interpretation. In fact it is only because of this conceptual independence that proofs of soundness and completeness, even the very notions of soundness and completeness, are conceptually coherent. In computer systems, however—and minds, and clocks—there is no reason to expect this total degree of disconnection or independence. We should expect something more like the relationship between the mass and velocity of a physical object, on the one hand, and the center of gravity or resonance of the system of which it is a part, on the other: a web of constraints and conditions tying the two factors together—piece-wise, incrementally—thereby giving rise to a comprehensive whole. The situation of a cmplete proof system defined on an abstract set of mathematical expression types is extreme: a global but locally unmediated coherence, with no part of the proof or inferential system touching the semantic interpretation or content, except in the final analysis, when an outside theorist's proof grandly ties the whole thing together. For computers, and for us, it seems much more plausible to take a step or two apart from our subject matter, and then check in with it, to stay in "synch"—by taking a look, for example, or (following AT&T's recommendation) by "reaching out and touching it." Participation is a resource, not a complication

Third, as both the first two points make clear, it is a little hard to justify calling the two factors *semantical*, especially when the first is shared with every other participant in the natural order. It is not just that the first should be viewed as syntax, the second as semantics (as application of this more general framework to the predicate calculus would suggest). Rather, it is not clear what, if anything, the terms 'syntax' and 'semantics' should mean in a context where the coupling between factors is so much richer and more complex than in the traditional idealised case—if indeed they mean anything at all. Clockworks are mechanisms that enable first-factor behaviour—that much seems innocuous enough; calling the momentum of a clock's pendulum *semantic* is more difficult. First and second factors are not distinct objects that somehow cooperate in engendering semantical activity; rather, one and the same causal constituents of a semantic system play both first and second factor roles.

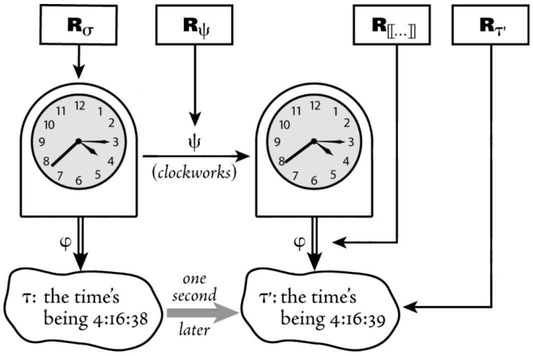


Figure 2 — The typology of clock semantics

This whole question is complicated by the use of the word 'semantics' (especially in AI) to describe inferential and structural relations among ingredients within a computational system. In (Smith, 1986)[x] I attempt to resolve some of these issues, but instead of reconstructing that argument here I will simply use the two-factor terminology without prejudice as to what does and does not have legitimate claim to the overloaded term.

5 Theoretic Machinery and Assumptions

Look, then, at how clocks represent time, starting with some basic assumptions. As suggested in figure 2, *qua* theorists we need accounts of four things:

1. States of the clock itself, including the face (σ);
2. The time or passage of time that the clock represents (τ);

[x] «Explain where that is done in this volume.»

3. The first factor movement or state change between clock states (ψ); and
4. The second factor representation relation (φ) between clock states and times.

All four of these are shared with standard semantical analysis: the first two would be the syntactic and semantic domain; the third, inference or proof theory; the fourth, semantics or interpretation.

I will adopt what I will call a **direct** rather than *model-theoretic* approach to these analytic tasks. Typically, when doing semantics, instead of talking directly about clock faces, orientations of hands. etc., one models them. For example, the state of a three-hand analog clock might be modelled as a triple, consisting of the orientations of the hour-hand, minute-hand, and second-hand, respectively, measured clock-wise from the vertical, in degrees. Thus the clock face shown in figure 2 would be modelled as follows:

$$M_\sigma\text{: <128.3166..., 99.8, 228>} \qquad \text{(S1)}$$

The problem with this technique, however, as suggested in figure 3, is that a model M of a situation S is itself a *representation* of S, since modelling is a particular species of representation (M_σ, for example, *represents* the clock face; it is not the clock face, since for example it has a length of three). The general character and complexity of the model–clock relation M_σ–σ, therefore, is the same as that between the clock and the time it represents (σ–t). It is therefore very hard to know whether what is crucial about σ–τ will be revealed or hidden if its analysis is conducted purely in a M_σ–M_τ form. For example, using simple numbers to represent the orientations of hands presumes an absolute accuracy on the clock face, counter to fact. When

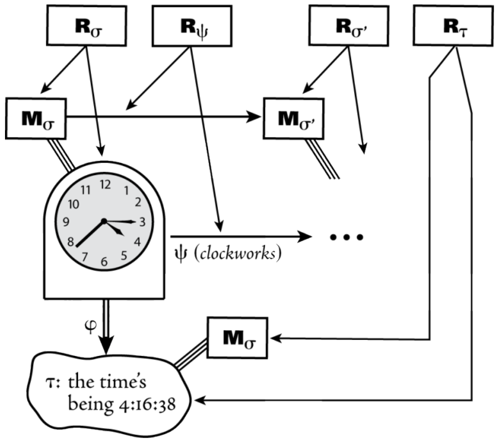


Figure 3 — The model-theoretic approach

studying something like natural language, which makes use of a much more complex representation relation than a model, the problems of indiscriminate theoretic modelling may be minor, or (more likely) go unnoticed. In our case, however, the representation relation under investigation—between clock faces and periodic time—is essentially an isomorphism. In this situation indiscriminate modelling would be theoretically distracting.

The direct semantical stance will have consequences, of two main sorts. First, we will need some machinery for talking precisely about the world without modelling it; for this I will use an informal "pocket situation theory," based unapologetically on Barwise and Perry.[8] Second, in the analog case it will be tempting to use some elementary calculus, which if I was going to do anything complex would be problematic, since a situation-theoretic reconstruction of continuity has not been yet been developed. On the other hand, since the continuities underlying the integrity of the calculus presumably derive, ultimately, from the fundamental continuity of the physical phenomena that the mathematics of the calculus was developed to describe, and since it is exactly such continuous phenomena that will be the subject matter here, I will take the liberty of applying its insights anyway. Since I will effectively merely be using mathematical notation, rather than actually doing any mathematics, this approach will not get us into trouble.

The direct semantical stance also highlights a question: how as theorists are we going to describe or **register**[x] the phenomena we are going to study—i.e., in terms of what concepts, categories, and constraints are we going to explicate its regularity? When giving semantical analyses of linguistic or syntactic objects (sentences, expression types, etc.), tradition provides standard registrations in terms of constituent terms, predicate letters, etc. Similarly, purely abstract objects are typically categorised in advance—in terms of a defining set of properties or relations. Clocks, on the other hand, are neither traditional nor abstract, so we have to address the question *de novo*, as it were.

My metaphysical bias is to treat the world as infinitely rich,

[8] Barwise and Perry 1983; Barwise, 1986a.

[x] «Ref "Rehabilitating Representation" for a discussion of registration.»

not only in the sense of taking there to exist more to everything than we can say, but also in assuming that there is both more uniformity and structure, and more heterogeneity and individual difference, than theory or language can ever encompass. I will therefore assume that clock faces, being actual, are sufficiently structured that one can be wrong about them, but still do not come labelled in advance by God, like plant slips at a nursery, identified with a white plastic tag with the name printed on them. Since every clock face, furthermore, exemplifies an infinite number of properties and relations (such as the property of being the subject matter of this paragraph), even after settling on a basic registration scheme, we have considerable latitude in making a specific choice.

None of this is intended to be either problematic or new; it is worth mentioning only because we need to make room for there being a difference between how we theorists do it, and how clocks do it, for themselves or (more likely, in the case of clocks) for their users. The problem is particularly acute for time itself, especially the periodic cycle of hours, minutes and seconds to which I keep referring without explanation. If this were a paper on the semantics of *time*, not just on the semantics of clocks, or even on the nature of time itself, not only would such an explication have to be given, but the incestuous fact addressed that clocks themselves are surely in part responsible for the temporal registration (hours, minutes, seconds, etc.) of the times they represent, as argued for example by Mumford (1934). In this paper, however, I will merely adopt the periodic cycle without analysis, taking its explanation as a debt that needs to be paid here.

Given these preliminaries, I summarise the ontological type structure that I will adopt in figure 4. Variables ranging over objects will be indicated with lower-case italic letters; over properties and relations, in lower-case Greek; over functions, in upper-case Greek. Thus *c* and *c'* will range over clocks; *t*, *t'*, etc., over full-blooded times, which are taken to be instantaneous slices through the metaphysical flux. Times are meant to include the time Kennedy was shot, the referent of 'now' (on any occasion of its use), the point when the ship passed out of sight behind the island—that sort of thing. Intervals—intuitively, temporal durations be-

Objects and Properties

$c, c', \ldots$	— clocks
$t, t', \ldots$	— times (instantaneous moments)
$\Delta t, \Delta t', \ldots$	— temporal intervals
$\tau, \tau', \ldots$	— o'clock properties (being midnight, being 4:01:23, …
	τ_t — the o'clock property that holds of time t
$\sigma, \sigma', \ldots$	— states of clock faces (both hands point upwards, …)
	$\sigma_{t,c}$ — the state of clock c at time t

Primary Theoretic Functions

$\psi: \sigma, \Delta t \rightarrow \sigma$	— clockworks (clocks states × intervals → clock states)
$\Sigma: c, t \rightarrow \sigma$	— state function (clocks × times → clock states)
$[\![\ldots]\!]: \sigma \rightarrow \tau$	— semantic content (clock states → o'clock properties)

Overloaded Addition

$t{+}\Delta t$: t	— times plus intervals are times
$\tau{+}\Delta t: \tau$	— o'clock properties + intervals are o'clock properties

Figure 4 — Theoretic type structure

tween times—will be indicated by Δt, $\Delta t'$, etc. I will extend the use of '+' to allow adding intervals to times (i.e., will "overloading '+'," as computer scientists would put it); thus $t{+}\Delta t'$ will be taken to be of the same type as t.

As opposed to times themselves being periodic (we will be more Heraclitean about them), I will assume that times are "located" on the periodic cycle by what I will call the **o'clock properties**—such as that of "being 4:01:23," "being midnight," etc. The idea is not so much to license a continuum of distinct properties, but rather to assume that these properties arise out of a continuous relation between times and the abstract locations on the periodic time cycle to which they are taken to correspond ("4:00," etc.). Various explanations of this relation are possible, but since the e intent of this paper is not to present an independently justified metaphysical account of time, but only to relate clocks to such a thing, I will employ a notation that simply picks up o'clock properties, whatever they are, from times that have them. Thus I will use τ_t to refer to the particular o'clock property that actually holds of time t. Also, I will take differences between o'clock properties to be intervals (e.g., the difference between 5:00 and 3:00 will be *two hours*). Thus the sentence $\tau_t(t')$ says of time t' that o'clock property $\tau_t(t)$—i.e., that it has whatever o'clock property t has. $\tau_t(t)$ is analytically true, therefore; as is $\tau_t(t+24{:}00{:}00)$. The term $\tau_t - \tau_{t'}$ denotes an interval, of type Δt.[9]

In an analogous way, σ, σ', etc. will range over a (continuous, in the analog case) set of states of clock faces. For traditional circular analog clocks, a σ representation 4:30 might be "having the hour hand at 135°, the minute hand at 180°, and the second hand at 0°, all measured clockwise from the 'XII.'

Given this framework, we can type the various semantical functions already encountered. As suggested in the previous section, Σ will be a (non-computed!) function of type $t \rightarrow \sigma$, from times onto clock states; ψ, a function of type $\sigma \times \Delta t \rightarrow \sigma$ from clock states and temporal intervals onto clocks states; and φ, a function of type $\sigma \rightarrow \tau$, from clock states onto o'clock properties. The important typological point for general semantic analysis is that both factors (ψ and φ) are defined as functions between the states that objects can be in, not between the objects that are in them. This is as you would expect for scientific laws.

Two more theoretical points, before we take up the analysis itself First, as just mentioned, I claimed in section 2 that times t were not causal agents—that they could not be in the domain of a strongly effective realisable function. It is probably more important to the life of clock designers that the o'clock properties (τ) are equally impotent. Even if it is 4:00 all around you, there is nothing that it's being 4:00 can cause to happen—such as serving tea and crumpets. With respect to engendering behaviour, a moment's being midnight is more like Boston's being a referent than it is like ice-cream's being sticky: it just is not the sort of thing that a sensor can or could detect. So functions of the form $\tau \rightarrow x$

[9] A more detached theoretic viewpoint should point out that o'clock properties τ are in fact two-place relations between times and places (a time that is midnight in London will be 7:00 p.m. in New York). More generally, whereas I assume throughout that activity (ψ) and interpretation (φ) are functions, they should properly be viewed as more complex relations between agents and their embedding circumstances.

are as non-realisable (in the strong sense discussed earlier) as those of type *t* → *x*, for arbitrary *x*. Such is life.

Second, I mentioned earlier that using numbers to represent the orientations of the hands of clocks presumes an accuracy that outstrips physical plausibility. Even if quantum physics would theoretically support there being a fact of the matter as to where a hand points within $\pm 10^{-50}$ degrees, say (which it will not), there are also pragmatic realities of producing a macroscopically observable clock subject to the forces of gravity, anomalies of manufacture, etc. Furthermore, if the hour-hand were anything like this accurate, then at least for theoretical purposes the minute and second hands would be redundant: a perfect observer could gaze at a clock and read off a time of, say, 4:15:38:17.[10] One might object, of course, that human users would not be able to register the hour-hand more accurately than, say, ±1° or ±2°, and therefore, even with internal calculation, would not be able to determine the time on a single-handed clock more accurately than to within about 5 minutes, no matter how much more accurately than that the time was actually signified. In fact casual observation suggests that, in reality, hour hands on modern analog clocks are caused, by the internal mechanism (clockworks) to be much more accurately positioned than is necessary merely to determine which hour the minute hand signifies time with respect to.

These issues again raise the question of the relation between how we as theorists register clock faces and the times they represent, and how clock faces themselves register those represented times.[11] But I will not answer this question here, since I will primarily be dealing with semantic constraints on clock and time registrations, rather than with individual registrations themselves.

6 Temporal Representation: The Second Factor

Given these premises and caveats, I turn to look at how times are represented. Intuitively, we are aiming for something like the fol-

[10] "Third, n…5. The sixtieth part of a second of time or arc."—*Webster's New International Dictionary,* Second Edition. New York: G. & C. Merriam, Co. 1934.

[11] Clock faces, and representations in general, do not need to register themselves, in order to represent.

lowing

$[\![$ $]\!]$ = the property of being 4:16 (S2)

To do this, we start with φ, of type $\sigma \rightarrow \tau$ from (representing) states of clock faces onto (represented) states of time—i.e., onto o'clock properties. Instead of the name 'φ', however, I will use so-called "semantic brackets" ('$[\![\ldots]\!]$') in the following way: $[\![\sigma_{c,t}]\!]$ will be the o'clock property signified by the state $\sigma_{c,t}$, where $\sigma_{c,t}$ is in turn taken to be the state σ of clock *c* at time *t*. For example, the sentence $[\![\sigma_{c,t'}]\!](t)$ claims of time *t* that it has the o'clock property that clock *c* indicates at time *t*'. Similarly, $[\![\psi(\sigma_{c,t'},\Delta t)]\!](t)$ claims of time *t* that has the o'clock property that clock *c* would (or did) indicate Δt later than time *t*', since $\psi(\sigma_{c,t'},\Delta t)$ indicates the state that it would be (or would have been) in then.

Using this terminology, we can say that clock *c* is chronologically **correct** at time *t* just in case *t* is of the type that the clock then indicates:

Correct(*c*, *t*) $\equiv_{\mathrm{df}} [\![\sigma_{\mathrm{c,t}}]\!](t)$ (S3)

So far, of course, this is a constraint on possible interpretation functions $[\![\ldots]\!]$, since I have not yet defined any specific instances. Longer-term notions of correctness (over extended intervals, for example) could be defined by quantifying over times; similarly, approximate degrees of correctness could be characterised in terms of the difference between what time it actually was and what time was indicated.

7 Clockwork: The First Factor

With respect to operation, the basic point is this: if at time *t* a clock is so-and-so (σ), then at some point Δt later it will be such-and-such (σ'), where $\sigma' = \psi(\sigma, \Delta t)$. The function ψ, which takes a clock into the future in this way, must be realised by the underlying physical machine—must be implemented, that is, by the clockworks. The important constraint on this relation, which I will call the **realisability** constraint, is that $\psi(\sigma, \Delta t)$ can depend on σ and on Δt, but not on the time *t* that is "happening" when the clock is in state σ.

In symbol manipulation or semantical contexts, where time and symbols are both digital, we often view ψ as a state-transition function (such as for a Turing machine controller). In such cases Δt drops out, being assumed to be a single temporal "click." For example, suppose S is a (discrete) function from states to states ($\sigma \rightarrow \sigma$). The equation for a single state change, of the sort one would expect in a digital world, would be something like $\sigma' = \mathrm{S}(\sigma)$—or, if generalized to Δt's of n ticks duration, $\sigma' = \mathrm{S}^{n}(\sigma)$. In the continuous world of physical mechanics, on the other hand, ψ is merely "what the world does," explained in terms of velocities, accelerations, etc. From this perspective, the calculus can be viewed as a theoretical vehicle with which to explain first factor futures for continuous systems, where the state σ of some system in an amount of time Δt after it is in a starting state σ_0, assumed. to depend on the continuity of the underlying phenomena, can be expressed in the familiar equation

$$\sigma = \sigma_0 + \frac{d\sigma}{dt}\Delta t + \frac{1}{2}\frac{d^2\sigma}{dt^2}\Delta t^2 + \ldots \qquad \text{(S4)}$$

My aim is not to contrast the discrete and continuous case (I want to develop results applicable to both analog and digital clocks), but rather to highlight the common focus on **state change**, represented computationally by state transition functions, and physically by temporal derivatives. There is, however, this apparent difference: the theoretic notions employed in physics (force, acceleration, etc.) are essentially "relative"; they describe how the new state will differ from the old one. The real identity of the new state—what state the system will actually arrive in—is obtained, as if it were conceptually subsidiary, by altering the previous state in the prescribed manner. State transition tables, in contrast, are typically "absolute." They still describe state *change*, of course—they are not temporal state functions like Σ. The point rather is that the new state is specific "*de novo,*" so to speak, not as a modification of the old one, though of course the extent to which the new state differs from the old can be calculated as a difference between the two.

This difference in theoretic stance, however, is superficial, since in actual use (in describing programs, operations on memory, etc.) state transition functions in computer science are almost always defined with explicit reference to how the new state differs from the old. In giving environment transition functions, for example, showing the consequence of binding a variable, the requisite function from total environments onto total environments is defined as modifying the value of the given variable in question, and *otherwise being just like the prior one*.[x] Practice suggests, in other words, that in the computational case, as in the physical case, state *change* is conceptually prior, new total state conceptually or ontically dependent. In both arenas, therefore—physics and computing—there is thus general support for my specific focus here on ψ.

Intuitively, a proper ψ for a clock will specify that it runs at the right speed. It is easy enough to calculate, in the case of circular analog clocks, that this amounts to having the hour hand, minute hand, and second hand rotate at 0.008333…°/sec, 0.1°/sec, and 6°/sec, respectively. But to *characterise* correctness this way is exactly like characterising the correctness of a proof procedure by pointing to the syntactic inference rules. It may indeed be true that, if this condition is is met, the clock will be running at the correct speed, but that does not mean that this condition expresses *what it is to be running correctly*. Rather, we want to say that if at time *t* (say, 12:00) a clock designates o'clock property $\tau_{t'}$ (say, 3:11), then at time $t+\Delta t$, (12:01, if Δt = one minute) it should indicate the o'clock property that would hold Δt later (i.e., 3:12). We can indicate this as follows:

$$\textbf{Right-speed}(\boldsymbol{c},\boldsymbol{t},\boldsymbol{\Delta t}) \equiv_{\mathrm{df}} [\![\sigma_{c,t+\Delta t}]\!] = [\![\sigma_{c,t}]\!]+\Delta t \qquad \text{(S5)}$$

which has the consequence, given the definition of ψ, that

$$[\![\psi(\sigma_{c,t+\Delta t})]\!] = [\![\sigma_{c,t}]\!]+\Delta t \qquad \text{(S6)}$$

Properly, it would probably be more pragmatically useful to state something stronger: that a clock runs correctly throughout the interval from *t* to $t+\Delta t$ if and only if it advances at the right speed for the whole time (note that the following is neutral as to whether this is a continuous or discrete interval—i.e., as to

[x] «Put in an explanation—maybe a sidebar?—on the "E/x→x′" notational abbreviation practice (even though the underlying formalism "requires" a total state designating function). This is a very curious—and telling—practice.»

whether ∀ is a discrete or continuous quantifier:

$$\textbf{Right-speed}(\boldsymbol{c},\boldsymbol{t},\boldsymbol{\Delta t}) \equiv_{\mathrm{df}} \quad \forall \Delta t' \mid 0 \leq t' \leq t \; \llbracket \sigma_{c,t+\Delta t} \rrbracket = \llbracket \sigma_{c,t} \rrbracket + \Delta t \tag{S7}$$

again directly yielding

$$\forall \Delta t' \mid 0 \leq t' \leq t \;\; \llbracket \psi(\sigma_{c,t+\Delta t}) \rrbracket = \llbracket \sigma_{c,t} \rrbracket + \Delta t \tag{S8}$$

These equations involve property identity, but I defer any questions on that issue to situation theory. Note also that in each version the two instances of '+' are of different types: the first takes a time and an interval onto a time; the second, an o'clock property and an interval onto a o'clock property. No problem.

Given (S3) and (S7), we can prove the temporal analogues of soundness and completeness: that if a clock is correct at time *t*, and runs at the right speed during the interval from *t* to *t'*, then it will be correct during that interval, and conversely if it is correct throughout the interval it must be running at the right speed. But it is more fun to do this in the continuous case, so let's turn to that.

Very simply, we want to talk of an analog clock's running at the right speed *instantaneously*, which means, intuitively, that we should differentiate the temporal state function Σ—or equivalently, take the limit of Σ as Δt approaches 0, in the standard way:

$$\underset{\Delta t \to 0}{\text{limit}} \frac{((\llbracket \sigma_{c,t} \rrbracket + \Delta t) - \llbracket \sigma_{c,t} \rrbracket)}{\Delta t} = \underset{\Delta t \to 0}{\text{limit}} \frac{(\llbracket \sigma_{c,t+\Delta t} \rrbracket - \llbracket \sigma_{c,t} \rrbracket)}{\Delta t} \tag{S9}$$

Since, as we have already said, differences between o'clock properties are intervals, the left side of this reduces to $\text{limit}_{\Delta t \to 0}(\Delta t/\Delta t)$, which is identically 1, yielding:

$$1 = \underset{\Delta t \to 0}{\text{limit}} \frac{(\llbracket \sigma_{c,t+\Delta t} \rrbracket - \llbracket \sigma_{c,t} \rrbracket)}{\Delta t} \tag{S10}$$

The right hand side, however, is merely the derivative, with respect to time, o the interpretation of the state. We cannot differentiate σ directly, its not being a function of time (in fact it is not a function at all), but we can rewrite (S10) in terms of Σ:

$$1 = \lim_{\Delta t \to 0} \frac{(\llbracket \Sigma(c,t+\Delta t) \rrbracket - \llbracket \Sigma(c,t) \rrbracket)}{\Delta t} \tag{S11}$$

This enables us to take the limit (Σ is continuous by assumption), since the right hand side is the derivative of a function that is essentially the composition of the second and first factors ($\varphi \circ \psi$, or equivalently and more applicably here, $\llbracket \ldots \rrbracket \circ \psi$).[12] I will abbreviate this as $\llbracket \Sigma \rrbracket$, giving us:

$$\textbf{Right-speed}_{\textbf{analog}}\boldsymbol{(c,t)} \equiv_{\mathrm{df}} \frac{\mathrm{d}}{\mathrm{d}t} \llbracket \Sigma \rrbracket = 1 \tag{S12}$$

If the derivative (with respect to time) of a function is unity, of course, it follows that the function is of the form $\lambda t \,.\, t+k$ for some constant k—or rather, in our case, $\lambda t \,.\, \tau_t+k$, as dictated by our type constraints, where k in this case is a constant of type Δt. This is exactly what we would expect; the constant represents the error in the clock's setting—the difference between the actual and indicated times . Predictably, the equation says if a clock is running at the right speed the error (the amount that it is "off") will remain (instantaneously) constant. Furthermore, since (S3) implies that

$$\text{Correct}(c,\, t) \text{ iff } \llbracket \Sigma(c,\, t) \rrbracket(t) \tag{S13}$$

it follows that the constant would be 0 for a correctly set clock, as expected.

We can summarise these results as follows:

$$\textbf{Correct}\boldsymbol{(c,t)} \equiv_{\mathrm{df}} \llbracket \Sigma(c,\, t) \rrbracket(t) \tag{S14}$$

$$\textbf{Right-speed}\boldsymbol{(c,t,\Delta t)} \equiv_{\mathrm{df}} \tag{S15}$$

$$\begin{array}{llll} & \forall \Delta t' \mid 0 \leq t' \leq t & \llbracket \sigma_{c,t+\Delta t} \rrbracket & = \llbracket \sigma_{c,t} \rrbracket + \Delta t \\ \text{implying that} & \forall \Delta t' \mid 0 \leq t' \leq t & \llbracket \psi(\sigma_{c,t+\Delta t}) \rrbracket & = \llbracket \sigma_{c,t} \rrbracket + \Delta t \\ \text{implying that} & \forall \Delta t' \mid 0 \leq t' \leq t & [\Sigma(c,t+\Delta t)] & = [\Sigma(c,t)] + \Delta t \end{array}$$

$$\textbf{Right-speed}_{\textbf{analog}}\boldsymbol{(c,t)} \equiv_{\mathrm{df}} \frac{\mathrm{d}}{\mathrm{d}t} \llbracket \Sigma \rrbracket = 1 \tag{S16}$$

and in their terms define what it is for a clock to be "working properly" from time t to $t+\Delta t$"

$$\textbf{Working}\boldsymbol{(c,t,\Delta t)} \equiv_{\mathrm{df}} \text{Correct}(c,t) \wedge \text{Right-speed}(c,t,\Delta t) \tag{S17}$$

[12] Strictly speaking this is not quite accurate, since both $\llbracket \ldots \rrbracket$ and Σ should depend on c and t: the function we are differentiating should really be $\lambda c,t \,.\, \llbracket \Sigma(c,t) \rrbracket$. But being strict would add only complexity, not insight.

$$\textbf{Working}_{\textbf{analog}}\boldsymbol{(c,t)} \equiv_{\text{df}} \text{Correct}(c,t) \wedge \text{Right-speed}_{\text{analog}}(c,t) \qquad \text{(S18)}$$

For either version, the constraint can be shown to be satisfied (over the interval, or instantaneously, depending) in exactly the following condition:

$$[\![\Sigma(c,t)]\!] = \lambda t \,.\, \tau_{\text{t}} \qquad \text{(S19)}$$

Given the abbreviation adopted above, we can state this even more simply:

$$[\![\Sigma]\!] = \lambda t \,.\, \tau_{\text{t}} \qquad \text{(S20)}$$

I would be the first to admit that (S20) is obvious—at least retroactively, in the scnse that, once stated, it is hard to imagine thinking anything else. In English, it says that the state function and the interpretation function should be proportional inverses: given a clock that (so to speak) maps time onto some sort of compelx motion, the appropriate interpretation function is merely that unction that maps that motion back on the o'clock properties of the linear progression of time that was started with. So the putative clock of figure 5, for example—with a million-mile pendulum and a 24-hour period—would have a pointer position (σ) proportional to sin(t), and an interpretation function analogously proportional to $\sin^{-1}(\sigma)$.[13]

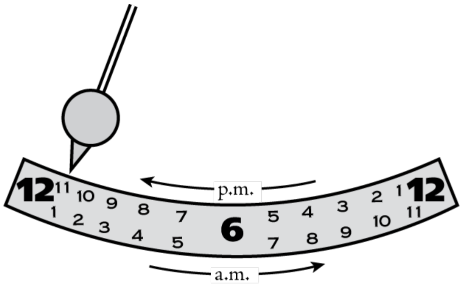


Figure 5 — The million-mile clock

Still, (S20) is not trivial, for a reason that shows exactly why clocks were hard to build. It says that working clocks *map all*

[13] This clock would be even harder to build than you might suppose. At first blush, it might seem as if the equation of motion for a pendulum would imply that a very large bob, swinging in an arc at the surface of the earth (an arc, say, 100 feet in length), whose mass completely dominated the mass of a long string by which it was suspended from an (energetically-maintained!) geosynchronous point 1,150,000 miles above the surface of the earth, would have a period of twenty-four hours. Unfortunately, however, such a device would have a period of slightly less than an hour and a half. Why this is so, and how to modify the design appropriately, are left as an exercise for the reader (hint: the result would be difficult to read).

times onto their o'clock properties. The problem for clockmakers is that Σ is not directly computable, since—to repeat—neither times nor o'clock properties enter into causally efficacious behaviour. What can be implemented is ψ, not Σ—and ψ is essentially the temporal derivative of Σ.

And that, in turn, leads us to the most compact characterisation of the function of clockworks:

> **The function of clockworks:** *to integrate the derivative of time*. To set the hands on the clock's face is *to supply the integration constant*.

8 Morals and Conclusions

What have we learned? Four things, other than some fun facts to tell our friends

The first has to do with the interaction among notions of *participation*, *realisation*, and *formality*. Clocks' participation in their subject matter (being temporal, as a way of measuring time), which depends on their physical realisation, might seem to violate the formality constraint that is claimed to hold of computational systems more generally. In fact, however, clocks' temporality does not relieve them of much of the structure that characterises more traditional systems: separable ψ and φ, the possibility of being wrong, etc. This similarity of clocks to symbol manipulation systems arises from the fact that the particular aspect of time that clocks represent—the o'clock properties—are not within immedate causal reach of a clockwork mechanism (or of much else, for that matter). In (Smith, forthcoming) I argue that this is a manifestation of a deep truth:

> *The limitations of causal reach are the real constraints on representational systems.*

Formality, as a notion, is merely a cloudy and approximate projection of these limitations into a particular construal of the symbolic realm.

The second moral has to do with the impact, for theoretical analysis, of the relations between ψ and φ. The function ψ, realised in clockwork, is what the engineers must implement; without an (explicit or tacit) understanding of it, functioning clocks could not be designed. The foregoing characterization of what it is for a clock to work properly, for example, had to reach beond the immediate or causally accessible aspects of the underlying clockwork mechanism. Whatever one might think about more complex cases, methodological solipsism does not work in this particular instance.

Third, the similarity between the state transition functions of computer science and the temporal derivatives of mechanics, both of which focus not on time itself but on temporal change, suggest the possibility of a more unified treatment of **representational dynamics** in general. So far most of what I have had to say has dealt with specific cases. So for example in section 2, I characterised inference as a particular species of representational activity having to do with changing content relations to a fixed subject matter, and contrasted it to a clock's maintenance of a fixed content relation to a changing subject matter. Remembering what is perceived is yet a different sort of representational behaviour: a form of retaining a fixed relation to a fixed subject matter, in ways that make it immune to changes in the agent's circumstances. And surely complex navigation in a busy world involves a dynamically-changing representational stance to a constantly-evolving situation. It does not seem impossible that a common framework could be uncovered

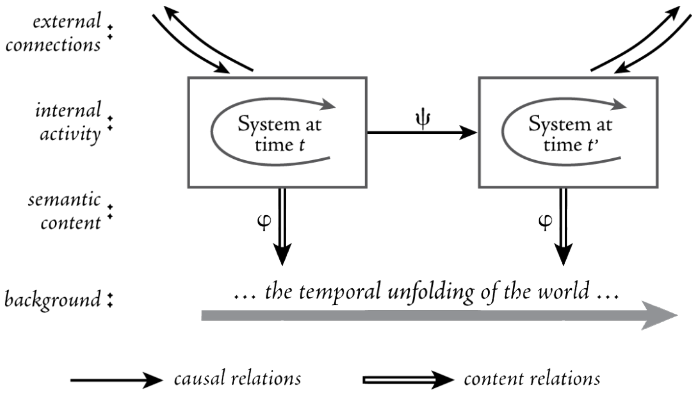


Figure 6 — C5: Coordinated Constraints on Content and Causal Connection

Fourth and finally, by occupying a place very different from that of either Turing machines or traditional theorem provers, clocks help illuminate the fundamental constraints governing computers and representational systems in general. As suggested in figure 6, there are two basic kinds of constraint—causal relations and content relations—that a representational system must coordinate as it moves through the world

Both kinds, in general, will be complex—much more so than we have seen in the case of clocks. Two aspects of content that I have not deal with here, for example, are its "situational" dependence on surrounding circumstances, as discussed for example in (Barwise 1986b and Perry 1986), and the three-way semantic interactions among language, mind, and world that arise in cases of communication. Causal connections are similarly complex, and can be broken down into three main groups:

1. **Internal activity of behaviour:** the relation between a system at some time and the same system shortly thereafter, which we called ψ;
2. **External connection:** Actions the system takes that affect the world, and effects on the system of the world around it—the results, that is, of sensors and effectors (clocks have none of this, but other systems are clearly not so limited); and
3. **Background dynamics:** The progress or flow of the surrounding situation—of which the passage of time would be counted as one instance, the behaviour of one's conversational partner, or a passing visual scene.

In the traditional case of pure mathematical inference, there is no connection (action or sensation), and the background situation, as we saw, is presumed to stay fixed. Barwise's particular construal of "formal inference"[14] strengthens this constraint by assuming that the content relation is also independent of surrounding situation. The clock examples give us a different point in the space: again no connection, an essentially unchanging (and relatively situation-independent) content relation, but an evolving background situation, mirrored in the internal activity or behaviour. Finally, semantic theories of action, involving everything from intentionally eating supper to making a promise, must deal with cases where the connection aspect makes a contribution. They must therefore deal with situations where the surrounding

[14] 'Formal' as meaning "non-situated"; see Barwise (1986b), p. 331.

situation is affected not only by background dynamics, but as a result of internal activity on the part of the representational agent. But simpler systems will require an analysis of external connection as well: computerised (ABS) automotive brakes systems, for example, are directly connected (even vulnerable) to the content of their representations, in a way that seems to free them from the need to have their representational states externally interpreted.

In the end, however, the similarity among these systems strikes me as far more important than the variance. I might put it this way. Causal participation in the world is ultimately a two-edged sword. On the one hand. it is absolutely enabling. Not only could a system not exist without it, but in a certain sense it is *total*: everything the system is and does arises out of Its causally supported existence. There are no angels. On the other hand, causal connection on its own—unless further structured—limits a system's total participation in the world to those things within immediate causal reach.

Representation, on this view, is a mechanism that honours the limits of causal participation, but at the same time stands a system in a content relation to aspeccts of the world beyond its causal reach. The trick that the system must solve is to live within the limits—and to exploit the freedoms!—of the causal laws in just such a way as to preserve its representational stance to what is distal. This much is in common between an inference system and a clock.

Acknowledgements

This paper grew out of a bet made with Richard Weyhrauch during a discussion late one night in a bar in Alghero, Sardinia, about what was involved in reading one's watch. Specifically, I promised to develop a semantical analysis of the familiar behaviour cited in the third paragraph of the paper: waiting a second to see whether a watch moved before reading the time. This paper is part one of the answer; interpreting a clock will come later. My thanks to him and other members of the Cost 13 Workshop on Reflection and Meta-Level Architectures, especially including Jim des Rivières and John Batali. Thanks also to Jon Barwise, John Etchemendy, David Israel, John Perry, Susan Stucky, and the other members of the situation theory and situation semantics (STASS) group at the Stanford Center for the Study of Language and Information (CSLI), to Pat Hayes for discussions of measurement, and to John Lamping for his help on celestial mechanics.

Supported in part by the Xerox Corporation and by the System Development Foundation, through their mutual support of CSLI; and for preparation for publication in 2009–10 by the Jackman Humanities Institute of the University of Toronto.

References

Barwise, Jon, and Perry, John: 1983, *Situations and Attitudes*, Bradford Books, Cambridge, MA.

Barwise, Jon: 1986a, "The Situation in Logic—III: Situations, Sets and <he Axiom of Foundation," in Alex Wilkie (ed.), *Logic Colloquium* 84, North Holland, Amsterdam Also available as CSLI Technical Report CSLI–85–26 from the Center for the Study of Language and Information, Stanford University, 1985.

Barwise, Jon: 1986b, "Information and Circumstance," *Notre Dame Journal of Formal Logic*, 27(3) 324–38.

Brachman, Ronald J., and Levesque, Hector J. (eds.): 1985, *Readings in Knowledge Representation* Morgan Kaufmann, Los Altos, CA.

Fodor, Jerry: 1975, *The Language of Thought*, Thomas Y. Crowell Co., New York. Paperback version, Harvard University Press Cambridge, MA, 1979.

Fodor, Jerry: 1980, "Methodological Solipsism Considered as a Research Strategy in Cognitive Psychology," *The Behavioral and Brain Sciences*, 3(1), pp. 63-73 Reprinted in Fodor, Jerry, *Representation*, Bradford, Cambridge, MA, 1981.

Mumford, Lewis: 1934, *Technics and Civilization*, Harcourt, Brace &Co., New York Reprinted 1943.

Newell, Allen: 1980, "Physical Symbol Systems," *Cognitive Science* 4, 135-183.

Perry, John: 1986, "Circumstantial Attitudes and Benevolent Cognition," J. Butterfield (ed.), *Language, Mind and Logic*, pp. 123–34, Cambridge University Press, Cambridge.

Postman, Neil: 1985, Amusing *Ourselves to Death: Public Discourse in the Age of Show Business*, Penguin Books, New York.

Smith, Brian Cantwell: 1982, *Reflection and Semantics in a Procedural* Language, Technical Report MIT/LCS 272, M.I.T., Cambridge, MA, 495 pp. Prologue reprinted in (Brachman and Levesque, 1985), pp.31-39.

———: 1984. "Reflection and Semantics in LISP," *Conference Record of the 11th Principles of Programming Languages Conference*, pp. 23–35, Salt Lake City, Utah, Also available as Xerox PARC Intelligent Systems Laboratory Technical Report ISL-5, Palo Alto, California, 1984.

———: 1986, "The Correspondence Continuum," appeared with the *Proceedings of the Sixth Canadian AI Conference*, Montreal, Canada, May 21-23. Available as CSLI Technical Report CSLI-87-71 from the Center for the Study of Language and Information, Stanford University, 1987.

——— *Is Computation Formal?* MIT *Press/A* Bradford Book Cambridge, MA. (Forthcoming.) «Explain»

Winograd, Terry, and Flores, Fernando: 1986, Understanding Computers and Cognition: A New Foundation for Design, Ablex, Norwood, NJ.

— Were this page been blank, that would have been unintentional —

D • Semantics

— Were this page been blank, that would have been unintentional —

9 — Linguistic & Computational Semantics[†]

Abstract

I argue that because the very concept of computation rests on notions of interpretation, the semantics of natural languages and the semantics of computational formalisms are in the deepest sense the same subject. The attempt to use computational formalisms in aid of an explanation of natural language semantics, therefore, is an enterprise that must be undertaken with particular care. I describe a framework for semantical analysis that I have used in the computational realm, and suggest that it may serve to underwrite computationally-oriented linguistic semantics as well. The major feature of this framework is the explicit recognition of both the declarative and the procedural import of meaningful expressions; I argue that whereas these two viewpoints have traditionally been taken as alternative, any comprehensive semantical theory must account for how both aspects of an expression contribute to its overall significance.

I have argued elsewhere[1] that the distinguishing mark of those objects and processes we call *computational* has to do with *attributed semantics:* we humans find computational processes coherent

[†] Slightly revised version of a paper that appeared in the *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics,* Toronto, Ontario, June 1982, pp. 9–15.

I am grateful to Barbara Grosz and Hector Levesque for their comments on an earlier draft of this short paper, and to Jane Robinson for her original suggestion that it be written.

[1] Smith (1982b).

exactly because we attach semantical significance to their behaviour. ingredients, and so forth. Put another way. computers, on this view. are those devices that we understand by *deploying our linguistic facilities.* For example. the reason that a calculator is a computer, but a car is not, is that we take the ingredients of the calculator to be *symbolic* (standing. in this particular case, for numbers and functions and so forth), and understand the interactions and organisation of the calculator in terms of that interpretation (this part *divides,* this part *represents the sum,* and so on). Even though by and large we are able to produce an explanation of the behaviour that does not rest on external semantic attribution (this is the *formality condition* mentioned by Fodor, Haugeland, and others[2]), we nonetheless speak, when we use computational terms, in terms of this semantics. These semantical concepts rest at the foundations of the discipline: the particular organisations that computers have—their computational *raison d'être*—emerge not only from their mechanical structure but also from their semantic interpretability. Similarly. the terms of art employed in computer science—*program, compiler. implementation, interpreter,* and so forth—will ultimately be definable only with reference to this attributed semantics; they will not, in my view, ever be found reducible to non-semantical predicates.[3]

This is a ramifying and problematic position, which I cannot defend here.[4] I may simply note, however, the overwhelming evidence in favour of a semantical approach manifested by everyday computational language. Even the simple view of computer science as the study of *symbol manipulation*[5] reveals this bias. Equally telling is the fact that programming languages are called *languages.*

[2] Fodor (1978), Fodor (1980), Haugeland (forthcoming)

[3] At least until the day arrives—if ever—when a successful psychology of language is presented wherein *all* of human semanticity is explained in non-semantical terms.

[4] Problematic because it defines computation in a manner that is derivative on mind (in that language is fundamentally a mental phenomenon), thus dashing the hope that computational psychology will offer a release from the semantic irreducibility of previous accounts of human cognition. Although I state this position and explore some of its consequences in Smith (1982b), a considerably fuller treatment will be provided in Smith (forthcoming).

[5] See for example Newell (1980).

In addition, language-derived concepts like *name* and *reference* and *semantics* permeate computational jargon (to say nothing of *interpreter, value, variable, memory, expression, identifier* and so on)—a fact that would be hard to explain if semantics were not crucially involved. It is not just that in discussing computation we *use* language; rather, in discussing computation we use words that suggest that we are also talking *about* linguistic phenomena.

The question I will focus on in this paper, very briefly, is this: if computational artefacts are fundamentally linguistic, and if, therefore. it is appropriate to analyse them in terms of formal theories of semantics (it is apparent that this is a widely held view), then what is the proper relationship between the so-called *computational* semantics that results, and more standard *linguistic* semantics (the discipline that studies people and their natural languages: how we mean, and what we are talking about. and all of. that good stuff)? And furthermore. what is it to *use* computational models to *explain* natural language semantics, if the computational models are themselves in need of semantical analysis? On the face of it, there would seem to be a certain complexity that should be sorted out.

In answering these questions I will argue approximately as follows: in the limit computational semantics and linguistic semantics will coincide, at least in underlying conception, if not in surface detail (for example some issues, like ambiguity, may arise in one case and not in the other). Unfortunately, however, as presently used in computer science the term 'semantics' is given such an operational cast that it distracts attention from the human attribution of significance to computational structures.[6] In contrast, the most successful models of natural language semantics. embodied for example in standard model theories and even in Mon-

[6] The term "semantics" is only one of a large collection of terms, unfortunately, that are technical terms in computer science and in the attendant cognitive disciplines (including logic, philosophy of language, linguistics, and psychology), with different meanings and different connotations. *Reference, interpretation, memory,* and *value* are just a few examples of the others. It is our view that in spite of the fact that semantical vocabulary is used in *different* ways, the fields are both semantical in fundamentally the *same* ways: a unification of terminology would only be for the best.

tague's program, have concentrated almost exclusively on *referential* or *denotational* aspects of declarative sentences. Judging only by surface use, in other words, computational semantics and linguistic semantics appear almost orthogonal in *concern*, even though they are of course similar in *style* (for example they both use meta-theoretic mathematical techniques—functional composition, and so forth—to recursively specify the semantics of complex expressions from a given set of primitive atoms and formation rules). It is striking, however, to observe two facts. First, computational semantics is being pushed (by people and by need) more and more towards declarative or referential issues. Second, natural language semantics, particularly in computationally-based studies, is focusing more and more on pragmatic questions of use and psychological import. Since computational linguistics operates under the computational hypothesis of mind, psychological issues are assumed to be modelled by a field of computational structures and the state of a processor running over them; thus these linguistic concerns with ;'use" connect naturally with the "operational" flavour of standard programming language semantics. It seems not implausible, therefore—I intend to betray caution with the double negative—that a unifying framework might be developed.

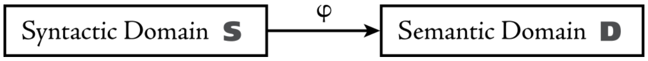


Figure 1 — Traditional (simple) semantical model

It will be the intent of this paper to present a specific, if preliminary, proposal for such a framework. First, however, some introductory comments. In a general sense of the term, *semantics* may be taken as the study of the relationship between entities or phenomena in a *syntactic domain* S and corresponding entities in a *semantic domain* S, as pictured in figure 1.

In accord with standard usage, I will call the function mapping elements from the first domain into elements of the second an **interpretation function** (to be sharply distinguished[7] from what in computer science is called an *interpreter*, which is a different beast altogether). Note that the question of whether an element is syn-

[7] An example of the phenomenon noted in footnote ■■.

tactic or semantic is a function of the point of view; the syntactic domain for one interpretation function can readily be the semantic domain of another (and a semantic domain may of course include its own syntactic domain).

Not all relationships, of course, count as semantical; the "grandmother" relationship fits into the picture just sketched, but stakes no claim on being semantical. Though it has often been discussed what constraints on such a relationship characterise genuinely semantical ones (compositionality or recursive specifiability, and a certain kind of formal character to the syntactic domain, are among those typically mentioned), I will not pursue such questions here. Rather, we I will complicate our diagram as indicated in figure 2, so as to enable us to characterise a rather large class of computational and linguistic formalisms:

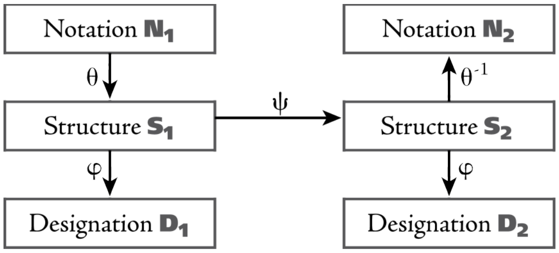


Figure 2 — Declarative & procedural semantical model

N_1 and N_2 are intended to be *notational* or *communicational expressions*, in some externally observable and consensually established medium of interaction, such as strings of characters, streams of words, or sequences of display images on a computer screen. The relationship θ is an interpretation function mapping notations into *internal elements* of some process over which the primary semantical and processing regimens are defined. In first-order logic, S_1 and S_2 would be something like abstract derivation tree types of first-order formulae; if the diagram were applied to the human mind, under the hypothesis of a formally encoded mentalese, S_1 and S_2 would be tokens of internal mentalese, and θ would be the function computed by the "linguistic" faculty (on a view such as that of Fodor[8]). In adopting these terms I mean to be speaking *very* generally; thus I mean to avoid, for example, any claim that *tokens* of English are *internalized* (a term I will use for

[8] Fodor (forthcoming)

θ) into recognisable mentalese tokens. In particular. the proper account of θ for humans could well simply describe how the field of mentalese structures, in some configuration, is transformed into some other configuration, upon being presented with a particular English sentence; this would still count, on the view I am presenting, as a theory of θ.

In contrast, φ is the interpretation function that makes explicit the standard denotational significance of linguistic terms, relating, we may presume, expressions in S to the world of discourse. The relationship between my mental token for T. S. Eliot, for example, and the poet himself, would be formulated as part of φ. Again, I am speaking very broadly; φ is intended to manifest what, paradigmatically, expressions are *about,* however that might best be formulated (φ includes for example the interpretation functions of standard model theories). ψ, in contrast, relates some internal structures or states to others—one can imagine it specifically as the formally computed derivability relationship in a logic ($\vdash$), as the function computed by the primitive language processor in a computational machine (i.e., as LISP's EVAL), or more generally as the function that relates one configuration of a field of symbols to another, in terms of the modifications engendered by some internal processor computing over those states. (φ and ψ are named, for mnemonic convenience, by analogy with *philosophy* and *psychology,* since a study of φ is a study of the relationship between expressions and the world—since philosophy takes you "out of your mind," so to speak—whereas a study of ψ is a study of the internal relationships between symbols, all of which, in contrast, are "within the head" of the person or machine.)

Some simple comments. First, N_1, N_2, S_1, S_2, D_1, and D_2 need not all necessarily be distinct: in a case where S_1 is a self-referential designator, for example, D_1 would be the same as S_1; similarly, in a case where ψ computed a function that was *designation-preserving,* then D_1 and D_2 would be identical. Secondly, we need not take a stand on which of φ and ψ has a prior claim to being the *semantics* of S_1. In standard logic, ψ (i.e., derivability: $\vdash$) is a relationship, but is far from a function, and there is little tendency to think of it as *semantical;* a study of ψ is called *proof* theory. In computational systems, on the other hand, ψ is typically much more constrained, and is also. by and large, analysed mathematically in terms of functions and so forth, in a manner much more like standard model theories. Although in my own view it seems a little far-fetched to call the internal relationships (the "use" of a symbol) *semantical*, it is nonetheless true that we are interested in characterising both, and it is unnecessary to express an a priori preference. For discussion, therefore, I will refer to the φ-semantics of a symbol or expression as its *declarative import*, and refer to its ψ-semantics as its *procedural consequence*. I have heard it said in other quarters that "procedural" and "declarative" theories of semantics are contenders;[9] to the extent that I have been able to make sense of these notions, it appears that we need both.

It is possible to use figure 2 to characterise a variety of standard formal systems. In the standard models of the λ-calculus. for example, the designation function φ takes λ-expressions onto functions; the procedural regimen ψ, usually consisting of α- and β-reductions, can be shown to be *φ-preserving*. Similarly, if in a standard predicate logic we take φ to be (the inverse of the) satisfaction relationship, with each element of S being a sentence or set of sentences, and elements of D being those possible worlds in which those sentences are true, and similarly take ψ as the derivability relationship, then soundness and completeness can be expressed as the equation $\psi(S_1, S_2) \equiv [D_1 \subseteq DS_2]$. As for all *formal systems* (these presumably subsume the computational ones), it is crucial that ψ be specifiable independent of φ. The λ-calculus and predicate logic systems, furthermore, have no notion of a processor with state; thus the appropriate ψ involves what we may call **local procedural consequence**, relating a simple symbol or set of symbols to another set. In a more complex computational circumstance, as I will show below, it is appropriate to characterise a more complex **full procedural consequence** involving not only simple expressions, but fuller encodings of the state of various aspects of the computational machine (for example. at least environments and continuations in the typical computational case[10]).

[9] Woods (1981)

[10] For a discussion of continuations see Gordon (1979), Steele and Sussman (1978), and Smith (1982a); the formal device is developed in Strachey & Wadsworth (1974).

An important consequence of the analysis illustrated in figure 2 is that it enables one to ask a question not typically asked ir computer science, about the φ-*semantic character* of the function computed by ψ. Note that questions about soundness and completeness in logic are exactly questions of this type. In separate research,[11] I have shown, by subjecting them to this kind of analysis, that computational formalisms can be usefully analysed in these terms as well. In particular, I demonstrated that the universally accepted LISP *evaluation* protocol is semantically confused, in the following sense: sometimes it *preserves* φ (i.e. $\varphi(\psi(S))=\varphi(S)$), and sometimes it *embodies* ϕ (thereby "de-referencing" its inputs: $\varphi(S)=\varphi(S)$. The traditional LISP notion of evaluation. in other words, conflates *simplification* and *reference* relationships, to its peril (in that report I propose some LISP dialects in which these two notions are kept much more neatly and strictly separate). The current moral, however. is merely that our approach allows the question of the semantical import of ψ to be asked.

As well as considering LISP, we can use our diagram to characterise various linguistically oriented projects carried on under the banner of "semantics." Model theories and formal theories of language (I am including Tarski and Montague in one sweep) have concentrated primarily on φ. Natural language semantics in some quarters[12] focuses on θ—i.e., on the "translation" of natural language into an internal medium—although the question of what aspects of a given sentence must be preserved in such a translation are of course of concern (no translator could ignore the salient properties, semantical and otherwise, of the target language, be it mentalese or predicate logic, since the endeavour would otherwise be without constraint). Lewis (for one) has argued that the project of articulating θ—an endeavour he calls "*markerese* semantics"—cannot really be called semantics at all,[13] since it is essentially a translation relationship, although it is worth noting that θ in computational formalisms is not always trivial, and a case can at least be made that many superficial aspects of natural language

[11] Smith (1982a).

[12] A classic example is Katz and Postal (1964), but much of the recent A.I. research in natural language in A.I. can be viewed in this light

[13] Lewis (1972)

use, such as the resolution of indexicals, may be resolved at this stage (if for example you say "*I am warm*" then I may internalise your use of the first person pronoun into *my* internal name for you).

Those artificial intelligence researchers working in knowledge representation, perhaps without too much distortion, can be divided into two groups: (i) those whose primary semantical allegiance is to φ, and who—perhaps as a consequence—typically use an encoding of first-order logic as their representation language; and (ii) those who concern themselves primarily with ψ, and who therefore—legitimately enough—reject logic as even suggestive (ψ in logic—derivability—is a relatively unconstrained relationship, for one thing; secondly, the relationship between the entailment relationship ($\models$), to which derivability is a hopeful approximation, and the proper ψ of rational belief revision, is at least a matter of debate.[14]

Programming language semantics, for reasons that can at least be explored, if not wholly explained, have focused primarily on ψ, although in ways that tend to confuse it with φ. Except in the case of PROLOG, which borrows its φ straight from a subset of first-order logic, and in my own reconstructions of the LISP, mentioned earlier,[15] I have never seen a semantical account of a programming language that gave *independent* accounts of φ and ψ. There are complexities, furthermore, in knowing just what the proper treatment of general languages should be. In a separate paper[16] I argue that the notion *program* is inherently defined as a set of expressions whose (φ-) semantic domain includes *data structures* (and set-theoretic entities built up over them). In other words, in a computational process that deals with finance, say, the *general* data structures will likely designate individuals and money and relationships among them, but the terms in that part of the process called a *program* will not designate these people and their money, but will instead designate *the data structures that designate people and money* (plus of course relationships and functions over

[14] Israel (1980).

[15] For a discussion of PROLOG see Oocksin & Mellish (1981); the LISPs are described in Smith (1981).

[16] Smith (forthcoming).

those data structures). Even on a *declarative* view like mine, in other words, the appropriate semantic domain for programs is built up over data structures—a situation strikingly like the standard semantical accounts that take abstract records or locations or whatever as elements of the otherwise mathematical domain for programming language semantics. It may be that this fact that all base terms in programs are *meta-syntactic* that has spawned the confusion between operations and reference in the computational setting.

Although the details of a general story remain to be worked out, the LISP case mentioned earlier is instructive, by way of suggestion as to how a more complete computational theory of language semantics might go.

In particular, because of the context relativity and non-local effects that can emerge from processing a LISP expression, ψ is not specifiable in a strict compositional way. When taken to include the broadest possible notion that maps entire configurations of the field of symbols and of the processor itself onto other configurations and states—ψ is of course recursively specifiable (the same fact, in essence, as saying that LISP is a deterministic formal calculus). A pure characterisation of ψ *without* a concomitant account of φ, however, is unmotivated—as empty as a specification of a derivability relationship would be for a calculus for which no semantics had been given. Of more interest is the ability to specify what I call a **general significance function** Σ, which recursively specifies ψ and φ together (this is what I was able to do for LISP). In particular, given any expression S_1, any configuration of the rest of the symbols, and any state of the processor, the function Σ will specify the configuration and state that would result (i.e., it will specify the *use* of S_1), and also the relationship to the world that the whole signifies. For example, given a LISP expression of the form `(+ 1 (PROG (SETQ A 2) A))`, Σ would specify that the whole expression designated the number three, that it would return the numeral '3', and that the machine would be left in a state in which the binding of the variable `A` was changed to the numeral '2'. A modest result; what is important is merely (i) that *both* declarative import and procedural significance must be reconstructed in order to tell the full story about LISP; and (ii) that they must be formulated together.

Rather than pursue this view in detail. it is helpful to set out several points that emerge from analyses developed within this framework:

1. In most programming languages, θ can be specified compositionally and independently of φ or ψ—this amounts to a formal statement of Fodor's *modularity* thesis for language.[17] In the case of *formal* systems, θ is often context-free and compositional, but not always (*reader macros* can render it opaque, or at least intensional, and some languages such as ALGOL are apparently context-sensitive). It is noteworthy. however. that there have been computational languages for which θ could not be specified independently of ψ—a fact that is often stated as the fact that the programming language "cannot be parsed except at runtime" (TECO and the first versions of SMALLTALK had this character).
2. Since LISP is computational, it follows that a *full* account of its ψ can be specified independent 0f φ; this is in essence the formality condition. It is important to bring out, however. that a *local* version of ψ will typically not be compositional in a modem computational formalism, even though such locality holds in purely extensional context-free side-effect free languages such as the λ-calculus.
3. It is widely agreed that ψ does not uniquely determine φ this is the "psychology narrowly construed" and the concomitant methodological solipsism of Putnam and Fodor and others[18]). However this fact is compatible with our foundational claim that computational systems are distinguished in virtue of having *some* version of φ as part of their characterisation. A very similar point can be made for logic: although any given logic can (presumably) be given a mathematically-specified model theory, that theory does

[17] Fodor (forthcoming).

[18] The term "methodological solipsism" is from Putnam (1975); see also Fodor (1980).

not typically tie down what is often called the "*standard model* or *interpretation*"—the interpretation that *we* use. This fact does not release us, however, from positing as a candidate logic only a formalism that humans can interpret.

4. The declarative interpretation function φ cannot be wholly determined independent of ψ, except in purely declarative languages (such as the λ-calculus and logic and so forth). This is to say that without some account of the effect on the processor of one fragment of a whole linguistic structure, it may be impossible to say what that processor will take another fragment as designating. The use of SETQ in LISP is an example; natural language instances will be explored below.

   This point needs a word of explanation. It is of course possible to specify φ in mathematical terms without any explicit mention of a ψ-like function; the approach I use in LISP defines both ψ and φ in terms of the overarching function Σ mentioned above, and I could of course simply define φ without defining ψ at all. Rather, my point is that any successful definition of φ will effectively *have to do the work of* ψ, more or less explicitly, either by defining some identifiable relationship, or else by embedding that relationship within the meta-theoretic machinery. I am arguing, in other words, only that the *subject* I intend ψ to cover must be treated in some fashion or other.

What is perhaps surprising about all of this machinery is that it must be brought to bear on a purely procedural language—*all three relationships* (θ, φ. and ψ) figure crucially in an account of even as simple a language as LISP. I are not suggesting that LISP is *like* natural languages. To point out just one crucial difference. there is no way in LISP or in any other programming language (except PROLOG) to *say* anything at all—whereas the ability to say things is clearly a foundational aspect of any human language. The problem in the procedural languages is one of what we may call **assertional force**; although it is possible to construct a sentence-like expression with a clear declarative semantics (such as some equivalent of "X=3"), one cannot use it in such a way as to actually mean it—so as to have it carry any assertional weight. That is, though it is trivial to *set* some variable X to 3. or to ask whether X is 3. there is no way to state *that* X *is* 3. It should be admitted, however, that computational languages bearing assertional force are under considerable current investigation. This general interest is probably one of the reasons for PROLOG's emergent popularity; other computational systems with an explicit declarative character include for example specification languages, data base models, constraint languages, and knowledge representation languages in Artificial Intelligence (AI). We can only assume that the appropriate semantics for all of these formalisms will align even more closely with an illuminating semantics for natural language.

What does all of this have to do with natural language, and with computational linguistics? The essential point is this: *if* this characterisation of formal systems is tenable, and if the techniques of standard programming language semantics can be fit into this mould. then it may be possible to combine those approaches with the techniques of programming language semantics and of logic and model theories, to construct complex and interacting accounts of ψ and of φ. To take just one example, the techniques that are used to construct mathematical accounts of environments and continuations might be brought to bear on the issue of dealing with the complex circumstances involving discourse models, theories of focus in dealing with anaphora, and so on; both cases involve an attempt to construct a recursively specifiable account of non-local interactions among disparate fragments of a composite text. But the contributions can proceed in the other direction as well: even from a very simple application of this framework to this circumstance of LISP, for example. we have been able to show how an accepted computational notion fails to cohere with our attributed linguistically based understanding, involving us in a major reconstruction of LISP's foundations. The similarities are striking.

My claim, in sum, is that similar phenomena occur in programming languages and natural languages, and that each discipline could benefit from the semantical techniques developed in the other. Some examples of these similar phenomena will help to motivate this view. The first is the issue with the appropriate use of noun phrases: as well as employing a noun phrase in a standard extensional (referential) position, natural language semantics has concerned itself with more difficult cases such as *intensional contexts* (as in the underlined designator in "*I didn't know that The Big Apple was an island*," where the co-designating term 'New York' cannot be substituted without changing the meaning), the so-called *attributive/referential* distinction of Donellan[19] (the difference, roughly, between using a noun phrase like "the man with a martini" to inform you that someone is drinking a martini, as opposed to a situation where one uses the hearer's belief or assumption that someone is drinking a martini to refer to him), and so on. Another example different from either of these is provided by the underlined term in "*For the next 20 years let's restrict the President's salary to $20,000*," on the reading in which after Reagan steps down he is allowed to earn as much as he pleases, but his successor comes under the constraint. The analogous computational cases include for example the use of an expression like (the formal analog of) "*make the sixth array element be 10*" (i.e., `A(6) ::= 10`), where we mean not that the current sixth element should be `10` (the current sixth array element might at the moment be `9`, and `9` cannot be `10`), but rather that we would like the description "the sixth array element" to refer to 10 (so-called "L-values," analogous to MacLISP's `SETF` construct). Or, to take a different case, suppose we say "*set `X` to the sixth array element*" (i.e., `x ::= A(6)`), where we mean not that `X` should be set to the current sixth array clement, but that it should *always* be equal to that element (stated computationally this might be phrased as saying that `X` should "track" `A(6)`; stated linguistically we might say that `X` should *mean* "the sixth array element"). Although this is not a standard type of assignment, the new constraint languages provide exactly such facilities, and macros (classic computational intensional operators) can be used in more traditional languages for such purposes. Or, for a final example, consider the standard declaration: `INTEGER X`, in which the term '`X`' refers neither to the variable *itself* (variables are *variables*, not numbers), nor to its current designation, but rather to *whatever will satisfy the description "the value of X" at any point in the course of a computation*. All in all, we cannot ignore the attempt on the

[19] Dannenan (1966).

we cannot ignore the attempt on the computationalists' part to provide complex mechanisms so strikingly similar to the complex ways we use noun phrases in English.

A very different sort of linguistic phenomenon that occurs in both programming languages and in natural language is what we might call “premature exits”: cases where the processing of a local fragment *aborts* the standard interpretation of an encompassing discourse. If for example I say to you “*I was walking down the street that leads to the house that Mary's aunt used to ... oh, forget it; I was taking a walk, and lo and behold...*”, then the fragment “forget it” must be understood as being used to discard the analysis of some amount of the previous clause. The grammatical structure of the subsequent phrase determines how much has been discarded, of course; the sentence would still be comprehensible if the phrase “an old house I like” followed the “forget it.” We are not accustomed to semantical theories that deal with phenomena like this, of course, but it is clear that any serious attempt to model real language understanding will have to face them. My present point is merely that continuations[20] enable computational formalisms to deal exactly with the computational analogs of this: so-called *escape operators* such as MacLISP's `THROW` and `CATCH` and `QUIT`.

In addition, a full semantics of language will want to deal with such sentences as “*If by “flustrated” you mean what I think, then she was certainly flustrated.*” The proper treatment of the first clause in this sentence will presumably involve lots of ψ-sorts of considerations: its contribution to the remainder of the sentence has more to do with the mental states of speaker and hearer than with the world being described by the presumed conversation. Once again, the overarching computational hypothesis suggests that the way these psychological effects must be modelled is in terms of alterations in the state of an internal process running over a field of computational structures,

As well as these specific examples, a couple of more general morals can be drawn, important in that they speak directly to styles of practice that we see in the literature. The first concerns the sug-

[20] See note 10 (■■), above.

gestion, apparently of some currency, that we reject the notion of logical form, and "do semantics directly" in a computational model On my account this is a mistake, pure and simple: to buy into the computational framework is to believe that the ingredients in any computational process are inherently linguistic, in need of interpretation. Thus they too will need semantics; the internalisation of English into a computer (θ) is a translation relationship (in the sense of preserving φ, presumably)—even if it is wildly contextual, and even if the internal language is very different in structure from the structure of English. It has sometimes been informally suggested, in an analogous vein, that Montague semantics cannot be taken seriously computationally, because the models that Montague proposes are "too big"—how could you possibly carry these infinite functions around in your head, we are asked to wonder. But of course this argument commits a use/mention mistake: the only valid computational reading of Montague would mean that mentalese (S) would consist of *designators* of the functions Montague proposes, and those designators can of course be a few short formulae,

It is another consequence of the view I am presenting that any semanticist who proposes some kind of "mental structure" in his or her account of language is committed to providing an interpretation of that structure. Consider for example a proposal that posits a notion of "focus" for a discourse fragment. Such a focus might be viewed as a (possibly abstract) entity in the world, or as a element of computational structure playing such-and-such role in the behavioural model of language understanding. It might seem that these are alternative accounts: what I am arguing is that an interpretation of the latter must give it a designation (φ); thus there would be a computational structure (being biased, I will call it the *focus-designator),* and a designation (which I will call the *focus-itself).* The complete account of focus would have to specify both of these (either directly, or else by relying on the generic declarative semantics to mediate between them), and also tell a story about how the focus-designator plays a causal role (ψ) in engendering the proper behaviour in the computational model of language understanding.

There is one final problem to be considered: what it is to design an internal formalism S (the task, we may presume, of anyone designing a knowledge representation language). Since, on my view, we must have a semantics, we have the option either of having the semantics informally described (or, even worse, tacitly assumed), or else we ean present an explicit account, either by defining such a story ourselves or by borrowing from someone else. If the LISP case can be taken as suggestive, a purely declarative model theory will be inadequate to handle the sorts of computational interactions that programming languages have required (and there is no *a priori* reason to assume that successful computational models for natural language will be found that are *simpler* than the programming languages the community has found necessary for the modest sorts of tasks computers are presently able to perform). However it is also reasonable to expect that no direct analog to programming language semantics will suffice, since they have to date been so concerned with purely procedural (behavioural) consequence. It seems at least reasonable to suppose that a general interpretation function, of the Σ sort mentioned earlier, may be required.

Consider for example the KLONE language presented by Brachman et al.[21] Although no semantics for KLONE has been presented, either procedural or declarative, its proponents have worked both in investigating the θ-semantics (how to translate English into KLONE), and in developing an informal account of the procedural aspects. Curiously, recent directions in that project would suggest that its authors expect to be able to provide a "declarative-only" account of KLONE semantics (i.e., expect to be able to present an account of φ independent of ψ), in spite of the foregoing remarks. My only comment is to remark that *independence* of procedural consequence is not a pre-requisite to an adequate semantics; the two can be recursively specifiable together; thus this apparent position is stronger than formally necessary—which makes it perhaps of considerable interest.

In sum, I claim that any semantical account of either natural language or computational language must specify θ, ψ, and φ; if any are left out, the account is not complete. I have argued, furthermore, that there is any fundamental distinction to be drawn be-

[21] Brachman (1979).

tween so-called procedural languages (of which LISP is the paradigmatic example in AI) and other more declarative languages (encodings of logic, or representation languages). I deny as well, contrary to at least some popular belief, the view that a mathematically well-specified semantics for a candidate "mentalese" must be satisfied by giving an independently specified *declarative* semantics (as would be possible for an encoding of logic, for example). The designers of KRL,[22] for example, for principled reasons, denied the possibility of giving a semantics independent of the procedures in which the KRL structures participated; my own simple account of LISP has at least suggested that such an approach could be pursued on a mathematically sound footing. Note however, in spite of my endorsement of what might be called a *procedural semantics,* that this in no way frees one from giving a declarative semantics as well; *procedural semantics* and *declarative semantics* are two pieces of a total story; they are not alternatives.

References

Bobrow, Daniel G., and Winograd, Terry, "An Overview of KRL: A Knowledge Representation Language", *Cognitive Science* 1 pp. 346, 1977.

Brachman, Ronald, "On the Epistemological Status of Semantic Networks", in Findler, Nicholas V. (ed.), *Associative Networks: Representation and Use of Knowledge by Computers,* New York: Academic Press, 1979.

Clocksin, W. F., and Mellish, C. S., *Programming in Prolog,* Berlin: Springer-Verlag, 1981.

Donnellan, K., "Reference and Definite Descriptions", *Philosophical Review* 75:3 (1966) pp. 281-304; reprinted in Rosenberg and Travis (eds.), *Readings in the Philosophy of Language,* PrenticeHall, 1971.

Fodor, Jerry, "Tom Swift and his Procedural Grandmother", *Cognition* 6, 1978; reprinted in Fodor (1981).

———, "Methodological Solipsism Considered as a Research Strategy in Cognitive Psychology", *The Behavioral and Brain Sciences.* 3:1 (1980) pp. 63-73: reprinted in Haugeland (cd.), *Mind Design,* Cambridge: Gradford, 1981, and in Fodor (1981).

[22] Bobrow and Winograd (1977).

Israel, David, "What's Wrong with Non-Monotonic Logic1", Proceedings of the First Annual Conference of the American Association for Artificial Intelligence, Stanford, California, 1980, pp. 99-101.

Katz. Jerrold, and Postal, Paul, *An Integrated Theory of Linguistic Descriptions*, Cambridge: M.LT. Press, 1964.

Lewis, David, "General Semantics", in Davidson and Harman (eds.), *Semantics 0/ Natural Langauges*, Dordrecht, Holland: D. Reidel, 1972, pp. 169-218.

Newell, Allen, "Physical Symbol Systems", *Cognitive Science* 4, pp. 135-183, 1980,

Putnam, Hilary, "The meaning of 'meaning'", in Putnam, Hilary, *Mind, Language and Reality*, Cambridge, U.K.: Cambridge University Press, 1975.

Smith, Brian Cantwell, *Reflection and Semantics in· a Procedural Language*, Laboratory for Computer Science Report LCS·TR·272, M.I.T., Cambridge, Mass., 1982 (a).

———, "Semantic Attribution and the Formality Condition", presented at the Eighth Annual Mecting of the Society for Philosophy and Psychology, London, Ontario, Canada, May 1316, 1982 (b). 15

———, *The Computational Metaphor*, Cambridge: Bradford (forthcoming).

Stcele, Guy, and Sussman, Gerald 1., "The Art of the Interpreter, or the Modularity Complex (parts Zero, One, and Two)", M.I.T. Artificial Intelligence Laboratory Memo AIM-453, Cambridge, Mass, 1978.

Strachey, C.. and Wadsworth, C. P., "Continuations—a Mathematical Semantics for Handling Full Jumps", PRG·H, Programming Research Group, University of Oxford, 1974.

Woods, William A., "Procedural Semantics as a Theory of Meaning", Report No. 4627, Bolt Beranek and Newman, 50 Moulton St, Cambridge, Mass., 02138; reprinted in Joshi, A., Sag, 1., and Webbcr, B., *Computational Aspects 0/ Linguistic Structures and Discourse Sel/ings*, Cambridge, U.K.: Cambridge University Press, 1982.

— Were this page been blank, that would have been unintentional —

10 — The Correspondence Continuum[†]

Abstract

It is argued that current techniques for analysing the semantics of knowledge representation systems in Artificial Intelligence (AI) are too rigid to account for the complexities of representational practice, and unable to explain intricate relations among *representation*, *specification*, *implementation*, *communication*, *modeling*, and *computation*. Doing justice to such phenomena challenges such stapes of traditional analysis as clear use/mention distinctions, strict metalanguage hierarchies, distinct "syntactic" and "semantic" accounts—even logic's notion of model-theory itself.

By way of alternative, the paper advocates the development of a **general theory of correspondence**, able to support an indefinite continuum of circumstantially dependent representation relations, ranging from fine-grained syntactic distinctions at the level of language and implementation, through functional data types, abstract models, and indirect classification, all the way to the represented situation in the real world. The overall structure and some general properties of such a correspondence theory are described, and its consequences for semantic analysis surveyed.

[†](A summary of) an earlier version of this paper was presented at the *Sixth Canadian Conference on Artificial Intelligence*, Montreal, Quebec, Canada, May 21–23, 1986,. While the paper did not appear in the *Proceedings* (having been submitted far too late), some copies were distributed at the meeting, and it was subsequently released as Report No. CSLI-87-71, Center for the Study of Language and Information, Stanford, California, in Jan. 1987.

Thanks to Jon Barwise, Jim des Rivieres, and John Lamping for comments on early (1986) draft. The research was supported by Xerox Corporation, and by a System Development Foundation award to the Center for the Study of Language and Information at Stanford University.

1 Introduction

Certain genitive phrases of the form '*a* of *b*' are ambiguous. On the *subjective* reading of 'love of children', for example, it is the children who do the loving, as in (1). On the *objective* reading, in contrast, children are recipients of the affection, as in (2).

1. Though bitter from years of being ridiculed by adults, the old man was grateful for the love of children.
2. Though increasingly impatient with his peers, the old man never lost his love of children.

The problem arises when the head noun phrase a ('love') signifies an asymmetric two-place relation, since it is then unclear which argument place is filled by the *b* term following 'of'. As shown in these examples, the distinction is generally clear-cut, with the intended reading selected by context (this is why it a question of ambiguity, not vagueness).

The phrase "the representation of knowledge" is of this ambiguous type. Oddly enough, though, it is not clear which reading is intended. Is knowledge being represented (objective), or is knowledge doing the representing (subjective)? Both interpretations seem reasonable. For example, suppose we build a medical artificial intelligence (AI) system called DOC using FKRL, our "favourite knowledge representation language." On the objective reading, the ingredient structures would be viewed as *representing DOC's knowledge*, presumably implying that a semantics for FKRL should map FKRL structures onto knowledge (or perhaps onto a set-theoretic model of it, such as a possible-world structure). On the *subjective* reading, in contrast, DOC's knowledge, embodied in FKRL structures, would *itself* be taken as representational. In this case semantic analysis would map the representational structures onto the states of affairs in the world that DOC knows about—states of affairs involving drugs, diseases, and diagnoses.

To add to the confusion, it is not even clear exactly what the difference between the two readings would come to, in the knowledge representation case. It seems that a possible world structure modelling belief might be the same as a structure modelling the states of affairs that the belief is about. And yet beliefs and worldly states of affairs are not the same: the former, for example,

ample, are psychological, the latter not (at least in general). Thus, whereas an erudite doctor might be said to possess great knowledge, it would be senseless to say that she possesses great states of affairs.

Some of the confusion has a simple source: *both* 'representation' and 'knowledge' designate asymmetric, relational notions. Furthermore, the two relations are of the same general type; they both characterise phenomena that are *about* something—phenomena that refer to the world, that have meaning or content. For example, to say that a series of marks on a page is a representation of Winston Churchill is to say that there is some relation between those marks and the late British Prime Minister. Similarly, to say that your lawyer's knowledge is faulty is to comment on the relation between what is going on inside the lawyer's head and what is going on outside. Because they are both based on an underlying (asymmetric) relation of content, representation and knowledge are considered to be *semantic* or *intentional* notions (other intentional notions include language, belief, model, theory, specification). But to say that is not to say very much, at least not yet. It certainly does not explain how representation and knowledge differ. Nor does it clarify our starting question of how, in the knowledge representation case, they are supposed to relate.

This paper will try to sort this all out. Specifically, taking semantics as the general enterprise of describing intentional phenomena, I will address the question of what it is to give a semantic analysis of a knowledge representation system. I.e., whereas most semantical analyses focus on *particular* types of semantic entities or structures—possible world structures, partial situations, etc.—my concern will be with the overall framework in terms of which such analyses are conducted.

There are several reasons this is an urgent task. The first we have already discussed: as implied by the confusion in the name, there are several interacting intentional notions involved, which should be sorted out. Second, it is increasingly thought necessary to give semantical accounts of proposed representation systems, in order to convey rigour and coherence onto what would otherwise be viewed as *ad hoc* symbol mongering. In 1974 Pat Hayes, long a champion of this view, called AI's reluctance to provide semantical accounts for representation schemes "a regrettable source of confusion and misunderstanding",[1] and went on in 1977 to write as follows:

> "One of the first task which faces a theory of representation is to give some account of what a representation or representational language *means*. Without such an account, comparisons between representations or languages can only be very superficial. Logical model theory provides such an analysis.[2]

In writing these words Hayes was defending logic against what he took to be the a-semantical orientation of the proceduralist tradition. In this he seems to have succeeded: similar sentiments have since gained widespread allegiance. We should certainly understand anything so popular.

On the other hand, this very success leads to the third reason for the present investigation. I believe that current theoretical tools, particularly the traditional model-theory that Hayes cites and most everyone uses, are inadequate to the knowledge representation task, and need substantial revamping. Perhaps ironically, many of the problems I will canvass are foreshadowed in Hayes' original papers—the relation between so-called propositional and analogue representation, to take just one example, which has yet to be adequately reconstructed. Logical model theory, which does not address analogue questions, has if anything gained momentum as the knowledge representer's semantical technique of choice.

Fourth, and finally, many of the lessons learned in the knowledge representation case will hold for all computational systems, and will even impinge on general semantical analysis; so there is a certain universality to the inquiry.

2 A Model of Knowledge Representation

I will adopt a **two-factor** model of knowledge representation, as pictured in figure 1 (on the next page). An agent, computational or human, is taken to comprise a set of internal structures, states, or aspects, that have some sort of content, and at the same time play

[1] Hayes, 1974 p. 64.
[2] Hayes 1977, pp. 559.

a role in engendering the agent's overall behaviour. In order to focus on their internal nature, I will call these structures **impressions**, to distinguish them from **expressions**, assumed to be elements of an external language. Think of impressions as data structures, as elements of a knowledge representation language, or as partial mental states—not much will depend here on details. The essential point about impressions is that they have two partially independent, though coordinated, properties.

First is what I will call **functional role** (or 'role', for short)—indicated as a in the diagram. Impressions must arise, somehow, in virtue of the history and coupling of the agent to its environment, and must give rise to the system's future activity or behaviour. Furthermore, as well as having these backwards- and forwards-looking aspects, impressions must be *causally efficacious in the present*—must bump up against each other, or be manipulated by some sort of internal agency, so as to constitute the whole of which they are the parts. So a given impression, such as one expressing the fact that a robot does not have much time left until it needs a recharge, might arise from the integration of information gleaned from internal sensor readings, engender inference involving time and expected electrical use, and lead the robot to scramble around the hall in search of an electrical outlet.

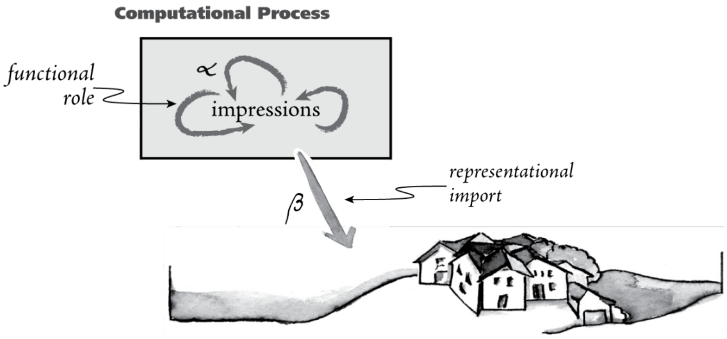


Figure 1 — A two-factor model of knowledge representation

Functional role is not enough, however. In order to count as representations, as opposed merely to being causal ingredients like the cam shaft in a car, impressions must also stand in a content relation to the states of affairs in the world in which the agent is embedded. I will call this second factor **representational import** (or just 'import', where the meaning is clear)—indicated in the diagram as b.

Representational import is not an alternative to functional role, or a particular kind: it is something additional. Thus whereas the level of sap in a maple tree arises from a complex history involving the weather, structure of the trunk, etc., and gives rise to complex future behaviour, such as amount of sugar produced, density of new foliage, etc., that is about all there is to say about it. In spite of being correlated with facts in its environment, sap does not have any representational import partly because the correlation is too strong (sap cannot be wrong), and partly because no concepts are employed (sap does not represent the world as *being one way as opposed to another*; it is merely locked into it as a totality). In contrast, for an impression to represent spring's being on the way, there would have to be an additional uniformity relating its structure to the structure of that fact—a uniformity that would be missed in an isolated account of functional role.[3]

For example, suppose I have the impression that water conducts electricity. All kinds of backwards-looking functional roles could have led to this: my own hapless experience trying to heat the bath water with an electric iron; stories I have been told; books I have read; deductions from knowledge of the ionization potential of molecules held together by hydrogen bonds. Similarly, at least within wide limits, there is no predicting what forward-looking role this impression might give rise to: things I might say, or situations I may strenuously avoid, such as climbing onto high-tension wires during rainstorms. The point is that, in spite of this richness of role, including inferential role, there remains a striking and relatively simple uniformity connecting the

[3] Saying just what distinguishes representational from purely functional ingredients is a difficult philosophical problem. My own emphasis on the two criteria cited here—a certain "disconnection" between representation and what is represented, and the claim that a representation must represent the world *as being a certain way*—is discussed in Smith ⟦forthcoming (a), chapter 4⟧, and in Smith ⟦forthcoming (b)⟧. The issue has been addressed by many writers in the philosophy of psychology, such as Fodor, Searle, and Stich, especially in assessing the relation between proposed functional and representational theories of mind. «Refs?» Computational readers will note, however, that many of these philosophers get at representation by analogy to computation, whereas my own view is approximately the opposite: that we must get at computation by first understanding representation. There is more overlap in subject matter than concurrence in views.

impression and the fact it represents—the most penetrating regularity in terms of which to explain my behaviour. In brief, it is the connection between the impression itself and *the fact that water conducts electricity*. This is the regularity of content or representational import.

The two factors must be coordinated in a special way: the states of affairs that the impression represents (its import) and the behaviour that it gives rise to (its role) must be such that the agent can be truthfully said to *know* the fact, which involves being able to act in accord with it, etc. The trick, in spelling this out, is to tie the two roles together into an integral whole without thereby undermining the integrity of the distinction between them—a project that requires combining traditional semantical techniques with the AI and philosophical literature on knowledge as action, pragmatic reasoning, and even causal theories of reference. I will not attempt that integration here, but will merely call the coordinated combination of factors the **full significance** of an impression.

In (Smith 1982a & 1985), I labelled this two factor orientation to representational significance the **Knowledge Representation Hypothesis**. In the philosophy of mind an analogous view has been labelled a *dual-component* semantics for psychology.[4] Technical variations have appeared under various descriptions; what is perhaps most striking is its familiarity in even the familiar realm of formal logic. In a traditional proof-theoretic framework—say, if the agent was an implementation of a natural deduction theorem-prover for first-order logic—one might view representational import as the *semantics* of an expression, and functional role as its *proof-theoretic* consequence. This last characterisation, however, misleadingly suggests that the full significance of a representation system must satisfy the following two constraints:

1. That the two factors be essentially *independent* (in which case I will call the representational system *declarative*); and
2. That functional role arise solely from *syntactic* properties of the representational structures.

Adherence to a general two-factor analysis, however, in no way

[4] Field (1977, 1978); Loar (1982); Block (1985).

commits one to this particularly strong way of dividing things up.[5] 3-Lisp, for example,[6] a simple programming language designed within a two-factor framework, explicitly violated both assumptions: import and role were both essentially semantic;[7] it was also shown that they were theoretically explicable only in intimate conjunction.[8] Other analyses, such as that suggested by Barwise and Perry,[9] propose alternative ways of tying content and behaviour together. In fact it is partly because there are so many ways of getting at roughly the same intuition that I have presented it here somewhat abstractly.

The two-factor nature of knowledge representation is the most important aspect for semantical analysis to clarify. In order to make sense of current semantical techniques, however, we need another distinction, which cross-cuts it.

Especially in the philosophical literature, semanticists sometimes distinguish the **meaning** of a structure from its **content** or **interpretation** (not, at least not in any straightforward way, to be confused with the computer science notion of interpretation; see section 5, below, and Smith (1984)). Very roughly, the former is what all instances or uses of a given structure type have in common; the latter, what a particular use or instance of that type re-

[5] David Israel has challenged the view, almost universally held in AI, that the notions of *proof, deduction, inference,* etc., even in mathematical logic, should be conceived in syntactic terms. «Refs» This syntactic orientation is not even universally accepted within what is called formal logic, since it rests on only one of many possible readings of the term 'formal' (see Smith ⟦forthcoming (a)⟧.

[6] Smith (1982a, 1984).

[7] Reasons why the functional (procedural) parts count as semantic are spelled out in Smith ⟦forthcoming (a)⟧.

[8] First factor derivability (£) and second factor satisfaction are traditionally tied together through entailment (|=) and proofs of soundness and completeness, but these particular notions are coherent only as a kind of global constraint on what are otherwise locally independent factors. The kind of "intimate conjunction" employed in 3-Lisp, and being imagined here for more general models of reasoning and computation, is one of much more local interdependence. As pointed out in Smith (1982b), computational practice already encompasses a wide range of such local interactions; see also Smith (1987).

[9] Barwise and Perry (1983).

fers to, or gets at, in all its specificity. Typically, facts about the context or setting provide the additional information that gets from meaning to interpretation. So for example the first person pronoun 'I', under this analysis, has the meaning of referring to whoever uses it: when Bono says 'I' he refers to himself; if you do, you refer to yourself. This is why two people can scream at each other "I'm right; you're wrong!"—they both use the same sentence, and *the meaning is constant*; it is the respective interpretations or contents that are contradictory. So we might model the meaning of 'I' as the following function of speakers, times, and locations as follows:

$$[\![\text{`I'}]\!] = \mathrm{l}s,t,l \cdot s$$

In a given situation of use (speaker s_o at time t_o in location p_o) the interpretation would thus be s_o.

It is tempting to identify meaning as the semantics of types, interpretation as the semantics of tokens—but the second of these is misleading. John Perry, for example, has imagined a case of two deaf mutes, so poor they must share a single tattered card saying *I'm a poor deaf mute; won't you give me some money.*[10] Standing together at the street corner, they alternately hand the card to passers by. Each time the card is used, the words 'I' and 'me' change their reference: one token, constant meaning, changing interpretation. Similarly, consider an analogous computational example: a machine with a single distinguished internal structure used to mean 'now'. The meaning is constant, and the particular structure may persist, but the interpretation changes with each passing nanosecond. Uses, or utterances, are what have interpretations; not concrete instances or tokens.

The meaning/interpretation vocabulary is not common in the AI or computer science literature, but the circumstantial dependence with which it deals is ubiquitous. Even the simple inclusion of explicit environment and memory arguments in denotational analyses of programming languages[11] manifests a sensitivity to the importance to interpretation of contextual factors. In Smith (1986) I layout a whole variety of ways in which the content of

[10] «Ref»

[11] See for example Gordon (1979).

computational structures, including impressions, can depend on facts of circumstance or context: internal facts (what program is running, how other internal structures are arranged, etc.), external facts (where the computer is located, whom it is conversing with, etc.), and even some facts that seem to cross the boundary (what time it is). The importance of these kinds of circumstantial dependence will be assumed in what follows.

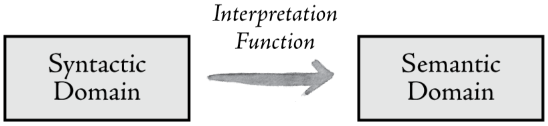


Figure 2 — The standard semantical model

Furthermore, both aspects of significance—functional role and representational import—can be circumstantially dependent. What ¬FLIES(*x*) means, when attached to the BIRD node in a default reasoning system, and what inferences it leads a system to draw, can both depend on the presence or absence of other intermediating impressions. I will use **functional meaning** and **representational meaning** to get at the respective factors of an impression's significance abstracted away from details and circumstances of particular instantiation or use. Similarly, **functional content** and **representational content** will refer, respectively, to the actions a use of an impression actually engenders, and to the situation it actually represents.

Given these distinctions, my overall question is this: *what would a semantical analysis be of the full significance of impressions?* In the broadest terms, it will clearly have to distinguish import and role, meaning and content, and show how they all come together into a coordinated whole. But we need details. I will proceed in steps, concentrating first on representational import. Later I will return to the question of how to tie it together with functional role.

3 The Present State

Virtually all the theoretical techniques in our current semantical arsenal were developed to deal with representational import. In particular, present practice proceeds roughly as suggested in figure 2. First, a source domain is identified as the set of elements for which a semantical analysis is to be given. Traditionally, this is called the **syntactic** domain; in the knowledge representation case it is the set of impressions comprising the agent (I will talk more about the difference in a moment). Second, a **semantic** domain is similarly identified, roughly taken to be what the elements of the representational domain, expressions or impressions, are about (more about 'aboutness', too, in a bit). Third, the semantic relation between domains, usually called the **interpretation function**, is then described *extensionally*, in the sense that particular elements of the syntactic domain are mapped, piecewise, onto the corresponding particular elements of the semantic domain. It may be, in the theorist's actual presentation of the semantic relation to the reader or audience or a colleague or whatever, that considerable information about the structure of this relation will be manifested, but strictly speaking this additional structure is not part of what is provided (or perhaps, to borrow from the *Tractatus*, we could say that it is *shown* but not *said*). Just as for functions and relations more generally, piecewise correspondence is assumed to be sufficient, at least for theoretic purposes.

So far, however, I have not said enough to distinguish the extensional analysis of a semantic relation from the extensional analysis of any old relation at all. But in practice more assumptions are adopted. I will label as **model-theoretic** those semantical analyses that accept (which I do not!) the following additional claims:

1. The elements of the representational domain are assumed to be *linguistic*. Debates rage over what language is, but at least the following seems agreed: complex linguistic elements are taken to be linear sequences of some sort (strings, utterances, whatever), with an inductively specified recursive structure founded on an initial base set of atomic elements called a **vocabulary**, and assembled according to rules of composition specified in a **grammar**. Furthermore, the interpretation relation is usually defined **compositionally**, so that meanings (not contents!) are assigned both to the vocabulary items and to the recursive structures engendered by the grammatical rules, in such a way that the meaning of a complex whole arises in a systematic way from the meanings of its parts. A particularly strong version of compositionality requires that the mean-

ing of a whole be definable, often by function composition, in terms of the meanings of the parts, but other possibilities, such as that the whole's meaning be characterised, or even just constrained, by systems of regularities among the parts, are growing in popularity. We need not take a position here on details; I will assume that these are *variants* on model-theoretic approaches.

2. In a case where the elements of syntactic domain *S* correspond to elements of semantic domain D_1, and the elements of D_1 are themselves linguistic, bearing their own interpretation relation to another semantic domain D_2, then the elements of the original domain *S* are called **metalinguistic**. Furthermore, the semantic relation is taken to be *non-transitive*, thereby embodying the idea of a strict use/mention distinction, and engendering the familiar hierarchy of *metalanguages*. This distinction is motivated by such obvious facts as that the six-character quoted expression " 'Nile' " designates a short word, which in turn designates a long river, but from those two facts it does not follow—nor is it true—that the original six-character expression " 'Nile' " itself designates the river.

3. The interpretation relation, as suggested in figure 2, is typically taken to be a *function*, implying that the import or content of an expression is not ambiguous. But ambiguity is a relative term: a linguistic element may look ambiguous if the circumstantial dependence of content has not been fully articulated, and may therefore be resolved by the meaning/interpretation distinction. We have already seen how the functional assumption is generalised to handle such complexities: whatever information disambiguates a given use of an otherwise ambiguous expression is included as a parameter of meaning; content is then obtained from the meaning by fixing that parameter. For example, the interpretation of the indexical expression 'I', discussed above, was parameterised on speakers (formally, for reasons to be explained in a moment, it was parameterized on speakers, times, and locations—though only the speaker affected the resulting interpretation). Similarly, if 'grue' means *blue if*

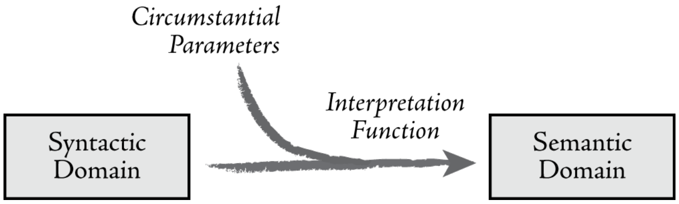


Figure 3 — Parameterised interpretation

used before some time t_0, and ‘green’ afterwards,[12] then its interpretation would be parameterized on time of use, leading to its being assigned roughly the following meaning:

$$[\![\text{‘grue’}]\!] = \text{l}s,t,l \cdot \text{if } t < t_0 \text{ then } \textsc{blue} \text{ else } \textsc{green}$$

Thus the true situation is more accurately pictured by figure 3, with dependence on circumstantial or contextual factors folded into the interpretation. As mentioned earlier, the discussion in Smith (1986) was intended to show how facts about both internal and external context can affect interpretation in this way.[13]

4. It is not necessary—not even usual—to require that the semantic domain D be the real domain that the expressions are about. Rather, D is required to be a set-theoretic structure, viewed as a **model** of the real semantic domain.

[12] See Goodman (1983).

[13] Functional parameterization deals with circumstantial dependence, but in a specific and limited way. In particular, by assuming that the linguistic element, plus circumstantial facts, together determine the interpretation, it implies that this is the direction of “information flow”—that understanding proceeds from knowledge of language, plus knowledge of circumstance, to knowledge of content. In practice, however, the flow can easily run in the other direction: someone hearing an utterance may know about the situation being described, and use that information to determine the structure of the linguistic element, or of such circumstantial factors as discourse structure. For these and other reasons a genuinely relational theory of meaning and content would be preferable (see Barwise and Perry (1983)); I use the functional analysis here only because of its familiarity, and because my current argument is not particularly sensitive to the distinction.

This last assumption serves a variety of useful functions: it means that semantical analysis remains "purely" mathematical, rather than having to spell out complete metaphysical assumptions about the true nature of the world. So for example a belief or proposition might be modelled as a function from possible worlds to truth-values, without the theorist needing to believe that that is what beliefs *really are* (but of course they are not functions in fact: it is entirely reasonable to ask "What are your friend's beliefs?", and absurd to ask "What are your friend's functions from possible worlds to truth-values?"). Similarly, in the semantical analysis of a language used to describe Turing machines, the semantical domain is usually taken to be sets of quadruples, not actual devices complete with tapes, read/write heads, finite state controllers, and so forth. The quadruples are viewed as a *model* of the Turing machine, and—this is the crucial point *modelling is assumed to be "free,"* in the sense that the theorist is granted license to engage in unconstrained modelling without having to account for it explicitly in his or her theory. To put it another way, modelling is invisible through the standard semanticist's glasses.[14]

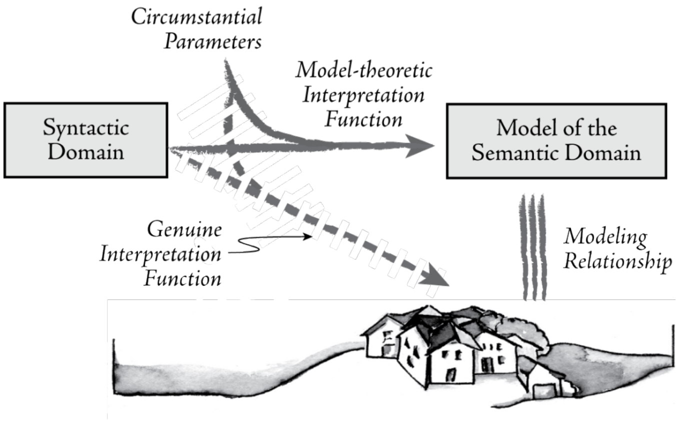

Circumstantial Parameters
Model-theoretic Interpretation Function
Syntactic Domain
Model of the Semantic Domain
Genuine Interpretation Function
Modeling Relationship

Figure 4 — Parameterised model-theoretic interpretation

Sometimes, of course, when the linguistic or representational elements are genuinely about mathematical objects—theories of arithmetic, for example, or representations of the factorial function—the true interpretation (called the 'intended interpretation') may be one of the model structures. In general, however,

[14] Sometimes, as for example in Montague semantics, the syntactic domain is modelled as well, but I will not worry about that here—it is merely an extension of the same points being made.

and almost universally in the knowledge representation case in artificial intelligence, we are interested in representations of more general states of affairs in the world, such as levels of digitalis in heart patients. So the picture of semantics should be updated as in figure 4.

Finally, in discussions to follow, we will encounter complex situations that include both modelling and iterated representation of the sort discussed in the second assumption. So it is important to summarise how the standard picture would look in such cases. Since modelling is typically ignored, such a situation would traditionally be *described* as a strict series of non-transitive denotation relations, each analysed piecewise. Our comments about modelling might suggest that the true situation is more complex, consisting of a series of non-transitive denotation relations, followed by an indefinite amount of promiscuous modelling. But in fact, since there may be promiscuous (i.e., invisible) modelling at each stage of the language hierarchy, as for example when a language is encoded in arithmetic (as is common in recursive function theory, for example), what we really have is this: a strictly non-transitive sequence, each step consisting of a denotation relation followed by an indefinite amount of promiscuous modelling. This situation is pictured in figure 5.[15]

[15] At least in this paper, I do not intend these remarks to challenge the appropriateness of these techniques for the intellectual project for which they were developed: the metamathematical inquiry into the foundations of mathematics. My current complaint is only about its adequacy for use in AI, knowledge representation, and any other situation in which the true state of affairs being represented is one in the real and messy world of everyday life.

4 Impressions

The first step, in analysing the appropriateness, for the representational problems presented in section 2, of the semantical techniques described in section 3, is to decide how we are going to treat *impressions*. Because I specifically introduced the term to cover any internal aspect, state (or partial state), or structure, we want a fairly general answer. It turns out to be a surprisingly complex subject. If we can clear it up first, therefore, subsequent semantical analysis will be that much more tractable.

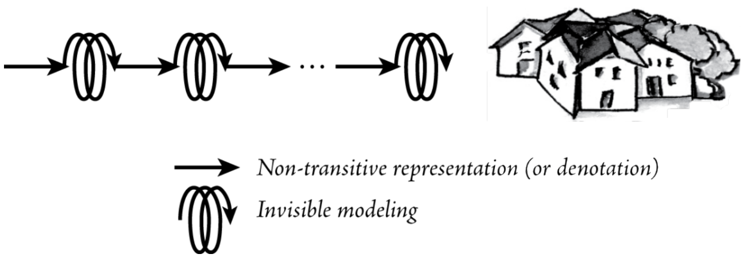


Figure 5 — Model-theoretic analysis of iterated representation

The most important point is this: *as semanticists*—whether our home field is in philosophy, artificial intelligence, logic, psychology, computer science, or artificial intelligence—*we do not yet have any developed theoretical terminology whose primary function is to describe impressions.* In particular, impressions are not necessarily *linguistic* objects, since the notion of language arises from the structure of communication and consensual interaction, not causal ingredients. Nor does mathematics provide any directly applicable notions: mathematical structures are abstractions—Platonic ideals, not fragments or constituents of activity. For example, in discussing two-factor semantical analysis in section 2, I talked about impressions being "causally efficacious"; these are not terms in the standard mathematical repertoire, nor, at least in general, are pure mathematical objects thought to possess causal powers. I have introduced the term 'impression' as a small step towards repairing this deficiency (as I did with 'structural field' in the 3-Lisp case), but of course it is simply a general, covering term. What we lack is a theory of types of impressions, types of important relations among impressions, analyses of how impressions can simultaneously cause and represent, and so forth. It is not that we are entirely without terms for such things: *data struc-*

tures, data bases, knowledge bases, data types, functions (in the 'procedure' sense), and *code* are all types of impression—as are more specific AI constructs such as *semantic nets, inheritance graphs,* and *taxonomic lattices.* Rather, what we need is a general theory, in terms of which these diverse kinds could be characterised.

Lacking a general theory, what do we theorists do instead? Different things. Perhaps the most common practice, especially in AI and the philosophy of mind, is to treat impressions metaphorically—in particular, as analogous to language. Thus in the cognitive case we have talk about "language of thought", "mentalese", "syntactic" theories of mind, etc.—as for example championed by Fodor, Stich, and others.[16] Artificial intellience typically follows the same path, talking about "expressions", knowledge representation "languages", etc.—as does anyone who views impressions as "formulae." In philosophy this stance is commonly referred to as the **representational theory of mind**—a somewhat unfortunate epithet, not because the term 'representation' is inherently so narrow,[17] but because this usage tends, without explicit admission, severely to constrain the notion of representation to its linguistic or even syntactic shadow. Instead I will call it a **linguistic theory of impressions**. Two facts about this theory are important for present purposes: (i) that we recognise its hypothetical nature—the fact that it represents a substantial claim; and (ii) that so long as this language remains metaphorical, we be careful to monitor connotations not necessarily warranted in the new domain.[18] For example, in 3-Lisp I called certain number-designating impressions *numerals,* but the metaphorical nature of the terminology misled me as well as others, causing me to attrib-

[16] See Fodor (1975), Stich (1985).

[17] See «whatever *Rehabilitating Representation* becomes».

[18] Boyd (1979) argues persuasively that metaphorical scientific language can play a role, especially initially, in enabling a community to establish increasingly substantial reference to a new domain. On such an account, the use of linguistic terminology to discuss impressions might, over the years, gradually lose its metaphorical overtones, and take on full-fledged referential connection to this new domain. But as Boyd himself points out, in order for this process to take hold, the metaphor must start out being at least partially correct. My concern in this particular case, as the rest of this section tries to suggest, is that many of the connotations of the use of linguistic language to describe impressions are in fact unwarranted..

ute semantical properties to impressions motivated more by linguistic connotation than by genuine functional need (for example, my adoption of a strict use/mention hierarchy, distinguishing the number three, the impression-numeral '3', and the expression-numeral "3").

Those bred in the knowledge representation tradition may find the linguistic approach to impressions obvious, but it is important to recognize that it is not universally accepted. It is well known that philosophical debates rage about whether representation is the best notion in terms of which to characterise human mental states. What is perhaps more surprising is the fact that a number of alternative views are advocated even within computational circles. First, many people have realised, in opposition to the linguistic claim in its narrowest form, that there is no need for internal structures to be anything like *identical* to written ones. The mildest position of this sort is John McCarthy's notion of "abstract syntax", which effectively amounts merely to a way to free impressions from gratuitous details of notation.[19] I made a stronger move in the same direction in developing 3-Lisp, using the term "**structural field**" for the totality of impressions, even though I then described individual impression types using terminology that I now feel was excessively derivative from linguistic analysis. My move was stronger than McCarthy's not only because the granularity of distinction in the 3-Lisp field was less than is usual in even abstract linguistic cases, but also because the mapping between expressions and impressions (as well as that between impressions that real world or tax-domain) was taken to be contextually sensitive. (Partly for reasons of circularity and structure-sharing, the external notation was neither isomorphic to internal impressions, nor complete. Furthermore, in certain complex cases like Lisp's closures, the impression structure was far more complex than linguistic notation could readily express.)

Other positions on impressions have been proposed. The view embodied in the design of 3-Lisp—that viewing impressions as syntactic or linguistic is non-ideal because it commits the theorist to too fine-grained a set of internal distinctions—was not mine alone; it is increasingly supported in various quarters of AI. Two

[19] «Ref»

suggested alternatives are of particular importance. Levesque (1984) retains allegiance to knowledge representation as a covering notion, but argues for a functional analysis of machine states, with explicit reference to the notion of an abstract data type, as opposed to a view of them as comprising "collections of symbolic structures."[20] Apparently more radically, Rosenschein (1985) criticises the entire representational stance, which he characterises as viewing "the state of the machine as encoding symbolic data objects"; Rosenschein argues instead for the notion of a situated automaton, with intentional properties (which he calls "knowledge") defined in terms of "objective correlations between machine states and world states".[21]

Supporting these anti-syntactical proposals, moreover, is the attitude towards impressions adopted in current theoretical computer science. Spelling that approach out is difficult, however, because of a facade of potentially distracting theoretical techniques that are standardly employed, which obscure (from the present vantage point) exactly what is going on. So I will digress from the subject of impressions, for a moment, to examine what computer sciences calls the denotational semantics of programming languages, and then return to the present topic once we have that firmly in hand.

5 Programs, Processes, and Indirect Classification

The abstract data type movement in programming language de-

[20] «Ref»

[21] History is often repeated, we are told, but here it is being repeated in reverse direction. The gradual shift from functionalism to representationalism in the philosophy of mind is apparently being played out backwards in AI, which started with a very strong representationalist stance, and is steadily moving away from it, towards what are explicitly admitted to be purely functional accounts (see Levesque (1984), Newell (1982), etc.). My own view is that both traditions, in opposite order, suffer from the lack of a full fledged theory of representation. Based on the idea that the only rigorous concept of representation is a narrow, purely syntactic version, they oscillate between its gratuitous detail and consequent semantic implausibility, on the one hand, and contextually insensitive and menacingly behaviourist pure functionalism, on the other. I believe both are inadequate, and conclude that we should free representation from its syntactic strictures, rather than rejecting the notion entirely.

sign, and the denotational approach to programming language semantics, are best understood as attempts to characterise the structure of computational processes in other than linguistic terms. They are motivated by the following obvious fact: when we develop computational processes, we *cannot build processes directly*. Instead, we cause them to come into existence by writing **programs**. In their discourse, AI programmers often gloss the distinction between the program and the process, viewing programs as functional ingredients that are either *inside* processes (a move in which programs are effectively taken to be impressions—partly motivated by the widespread use of interpreted, interactive languages like Lisp), or sit in the background causing them to exist, etc. Such assumptions are betrayed in such informal parlance as "*The program is still running*", "*The program reads in a number and then prints out the answer*", etc.

Nonetheless, as every programmer knows full well, *programs*—textual objects that are printed out on paper or on the screen, that are edited with EMACS and other editors, etc.—do not *do* anything; they are inert. Rather, what happens is that these passive structures are used by interpreters and compilers (about which more in a minute) to engender behaviour with appropriate properties.

The situation is depicted in figure 6. As just stated, the AI and knowledge representation community typically views programs, along with elements of knowledge representation languages, as constituents of or elements within computational processes—i.e., as impressions. I will call this the **ingrediential** view, as suggested in figure 6b. By far the more standard computer science conception, in contrast, is what I will call the **specificational** view, pictured in 6a: programs taken as specifications or descriptions of computations, albeit as special descriptions that can be viewed as *prescriptions* by the machine or interpreter. Different from both is a third, **conversational**, view, in which programs constitute the dialog or discourse that the programmer has with the machine—a view that I will examine in later, in section 6.

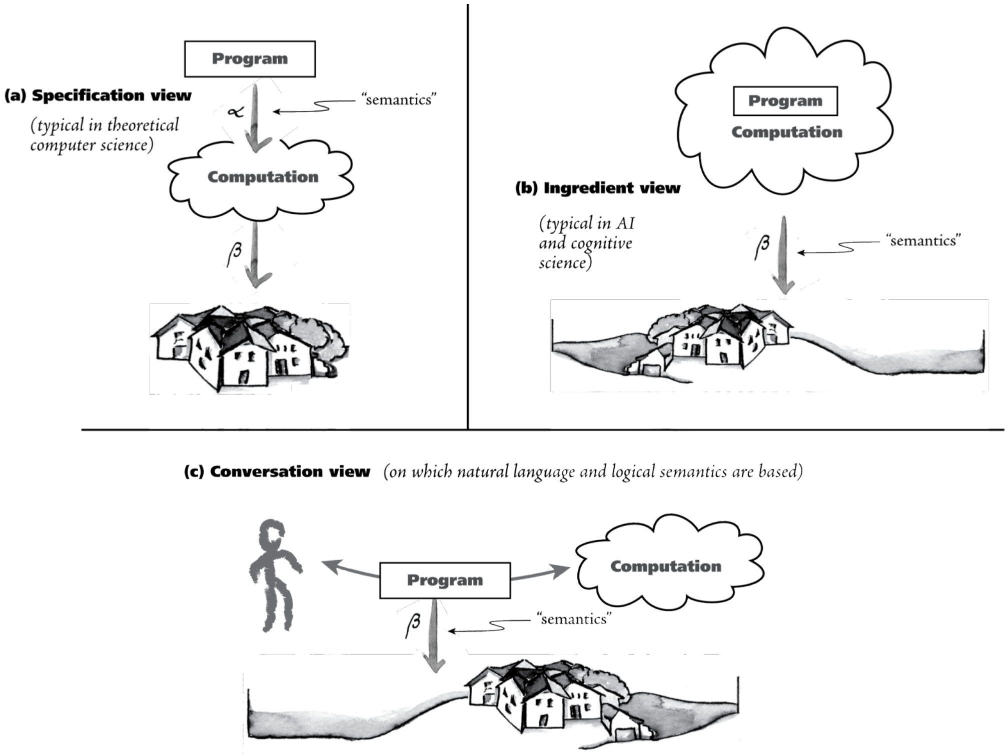

(a) Specification view
(typical in theoretical computer science)
Program
α
"semantics"
Computation
β
Program
Computation
(b) Ingredient view
(typical in AI and cognitive science)
β
"semantics"
(c) Conversation view (on which natural language and logical semantics are based)
Program
Computation
β
"semantics"

Figure 6 — Three Perspectives on Programs

The hugely important point, which will greatly affect our semantic analysis, is the following: *traditional computer science takes "semantics" to be the job of mapping programs onto processes*—not, as external observers, philosophers, linguists, etc., might expect, to be that of mapping the resulting processes onto the world. It is only under this C.S. conception, furthermore, that "interpreters" are properly named.

Concerned as I am in this paper, with knowledge representation, my task is different: exactly to describe that relation with which computer science does not concern itself—*between those (resulting) processes and the worlds in which they are embedded*. It follows that, in the traditional terminology, the *semantic* domains of traditional programming language analyses should be the knowledge representer's so-called *syntactic* domains. Confusion over this point amounts to the commission of a use/mention error—exactly the sort of thing that careful semantical analysis is so much at pains to eliminate.[22]

It may seem odd to look for impressions in the semantic domain of a semantic analysis of a programming language. Denotational semanticists, after all, typically deal in semantic domains consisting of abstract mathematical structures—functions, sets, numbers, partial orders, and the like—which do not seem very much like causally efficacious impressions. But this apparent discrepancy is explained by the fact that traditional denotational semantics is model-theoretic. As we have already seen, the model is not the true domain of interpretation, but some other structure, typically abstract, set in correspondence with it. As suggested earlier, this technique enables theorists at least partially to avoid exactly the metaphysical questions we are interested in: questions about the true nature of impressions themselves.[23]

[22] Although I will eventually challenge the idea of a rigid use/mention distinction, that does not mean that many so-called "use-mention confusions," such as this, are not serious..

[23] Some readers will object that computer science analyses treat computational processes only in terms of surface behaviour—input/output relations without positing any internal structure at all, let alone impressions. But this is not so clear, not only because I have defined impression in a rather general way, but also because this view assumes a purely "extensional" reading of the semantical analyses themselves. As has been argued by Fodor and others in the mental case, some sort of representational in-

Not all questions are avoided by employing model-theoretic strategies, of course, since the structure of the model is intended in some way to correspond to the structure of the impressions. The question is how the correspondence goes (i.e., what is the relation between a set-theoretic structure and an FKRL impression?). To get at the answer, note that modelling is an instance of the rather general practice of describing a set of complex phenomena only by setting them in relation to another, presumably more familiar, set of structures. Barwise and Perry call this "**indirect classification**".[24] An observer establishes (perhaps implicitly) a relation between the domain in question and some other domain, and then describes particular phenomena in the first domain only with reference to some corresponding phenomena in the second.

An obvious case, important to our present subject matter, is the folk classification of people's thoughts and beliefs: we describe what a person *P* believes by describing the situation that would be the case if what *P* believes were true. When you ask me to describe my thought, there is a perspective from which I am literally incapable of answering, since in English we have neither vocabulary nor intuitions about the direct structure of thoughts—i.e., about what is inside our minds, which is where most people would say thoughts lie. Rather, I am liable to say something like the following: "I was thinking *that Palo Alto is too far from Finland*." That is, I describe my thought or thought process indirectly, by adverting to a fact (Palo Alto's being too far from Finland) that would be the case if my thought were true. The examples we looked at in discussing model-theoretic semantics were just like this: the general practice is to establish an association between something and something else, and then to get at the something else by referring to the something. So for example we set up a correspondence between Turing machine states and quadruples, which lets us describe a particular state by referring to a par-

gredients will often be posited by theory merely in order to state the proper behavioural regularities. The abstract data types of denotational analysis can be viewed purely as theoretic entities, without classificatory import, but an argument would have to be made that they do not represent impression structure; the mere fact that they re not *claimed* to do so is not sufficient.

[24] Barwise and Perry (1983).

ticular quadruple.

These examples illustrate an important general property of all indirect classification: what is *specific* about a given entity in the primary domain is set in correspondence with what is *specific* about the corresponding entity in the classificatory domain. Thus a theoretical computer need not encode, in the domain of quadruples, the fact that Turing machines have tapes, or that the third element of the quadruple corresponds to the mark under the read/write head, or that the numbers 0 and 1 are used to classify a mark or a blank, or *anything else that is true for all the relevant cases.* All that is required is that a *particular* quadruple contain enough information to determine what *particular* state, transition, etc., that it is being used to classify.

What distinguishes the denotational approach to programming language semantics from arbitrary indirect classification, and leads to potential confusion, is the practice of *identifying* the classificatory entity with what is thereby classified. Such identification is not necessary; one could classify Turing machine #23 with quadruple #1437 without going on to claim that Turing machine #23 *is* quadruple #1437 (or even, more strongly, that to *be* a Turing machine is to *be* a quadruple—which of course is in fact false). The identification is considered to be acceptable when the two structures are thought isomorphic, but *isomorphism is always relative to an assumed metric of equivalence.* In the computational cases we are concerned with, where a second semantical factor (functional role) lurks in the background, in need of explanation, we cannot afford to identify for one purpose, two things that may differ in respects that matter for other purposes. In particular, two structures that look to be isomorphic from the point of view of representational import may differ, crucially, in terms of functional role. For example, as we have already pointed out, no abstract mathematical structure is even a candidate for the kind of efficacious causality we will need in order to connect impressions with action. Distinct but isomorphic mathematical structures may be used to classify embodied mechanisms with very different causal powers. So we need to proceed extremely cautiously.

We will encounter further issues about modelling in the next section, but for now let me return to programming languages.

In spite of its being contrary to the dominant view in AI and cognitive science, in what follows I will informally adopt the *specificational* view of programs, since it provides the most freedom, is least biased with respect to impression structure, and is most compatible with current computational theory. Thus I will assume: first, that programs are inert linguistic entities, built up of expressions; second, that, in contrast, processes are active, manifest behaviour, composed in part of causally-effective impressions; and third—which is where the specificational perspective takes hold—that denotational semantics in computer science is an analysis of the program-process relation that *indirectly classifies computational processes in terms of abstract mathematical models*. The situation is pictured in figure 7.

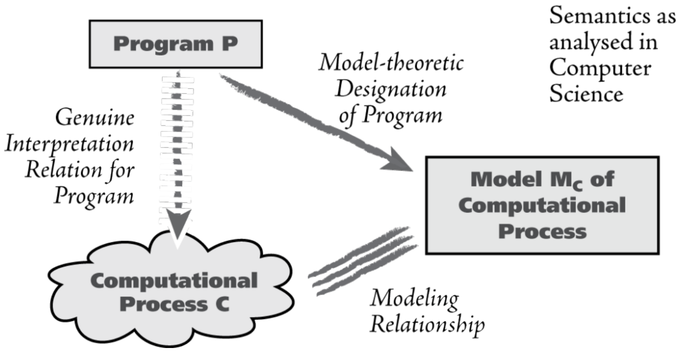

Program P
Genuine Interpretation Relation for Program
Model-theoretic Designation of Program
Semantics as analysed in Computer Science
Model M_C of Computational Process
Computational Process C
Modeling Relationship

Figure 7 — Model-theoretic analysis of program semantics

In terms of this picture, I can now explain the theoretical distraction I alluded to earlier, in introducing this section. It arises from the combination of two problems: (i) failing to distinguish between the specificational and ingrediential views of programs; and (ii) being seduced by model-theoretic properties of the model (its abstract, mathematical character) into thinking it must model content. The result is to lead one to identify the model M_c of the computational process C with the model M_w of the state of affairs *W* that the process is genuinely about—as shown in figure 8.

The fact that the programming language tradition calls its analyses *semantical,* in other words, coupled with the fact that they it tends to use abstract domains for purposes of indirect classification, is liable to mislead AI researchers into thinking that the semantic domains of programming languages *model the content of the computational processes that the programs engender*. But this is false, at least in general. *There is simply no assumption, in the standard semantical analysis of programming languages, that computational processes are themselves semantic or intentional entities.* That is, no *further* semantic relation is required, acknowledged, or described. All that is explained is the relation between program and engendered computational process.

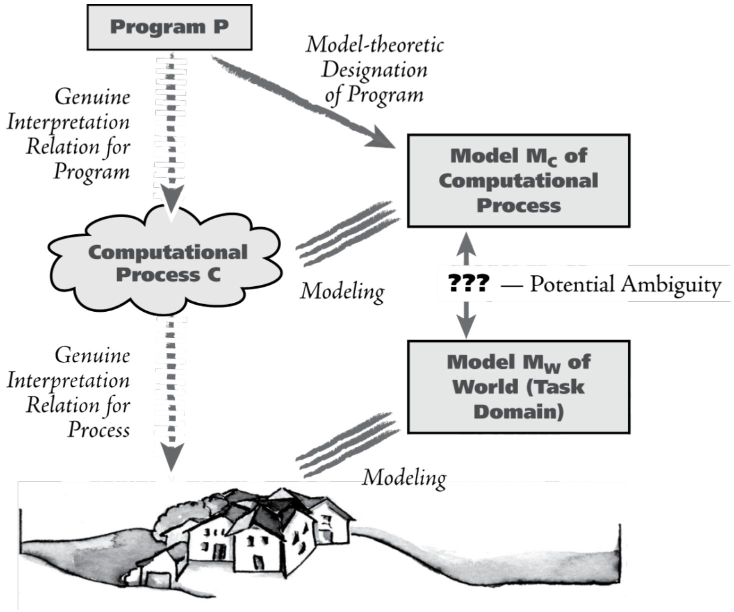

Program P
Model-theoretic Designation of Program
Genuine Interpretation Relation for Program
Model M_C of Computational Process
Computational Process C
Modeling
??? — Potential Ambiguity
Model M_W of World (Task Domain)
Genuine Interpretation Relation for Process
Modeling

Figure 8 — Models vs. Interpretations of Processes

In the AI case, however, and particularly when dealing with knowledge representation systems, we assume that the ingredients inside the processes we are interested in, which we are calling impressions, are *themselves* intentional (this was the essence of our adopting a representational, as opposed to a merely functional, stance in section 2). Even if we were to adopt a model-theoretic approach in our semantical task, therefore, we would be interested in the relation between impressions in *C* (or in the model M_c) and the model M_w.

I have already said that there is no *a priori* reason to assume that these two models M_c and M_w will be the same. But a stronger thing can be said: if one assumes that M_c is an adequate model of process C, and that M_w is an adequate model of what C is about, then

> *To identify Mc and Mw is to assume that the representational import relation of knowledge representation systems is one of isomorphism.*

Far from treating impressions as a *language*, this would be to treat them as a *simulacrum of the world*. Or to put the same point another way: to adopt, as a model of a knowledge representation system's semantics, a denotational analysis of the programming language used to specify it, is either to assume that the primary representation relation, between process and world, is one of isomorphism, or else—even worse—to ignore that relation completely (thereby maintaining a solipsistic stance towards computations themselves). Either result is unhappy: simultaneously false and terrifically misleading.

It helps to look at some examples, starting very simply.

In purely mathematical cases, as mentioned, M_c and M_w may truly coincide. For example, suppose we write a program to calculate the factorial function. We may presume this literally means the following: that we write a program to specify a process that is about numbers and the factorial relation. In this case W is a structured domain of numbers and functions. Moreover, a denotational semanticist in computer science would almost surely use the same structures (numbers and the factorial function) as an abstract mathematical model (M_c) in terms of which to classify the process. Not only can M_c and M_w be identified, in other words; in this situation M_c, M_w, and $W$are identical.

As is perfectly evident, however, this identity relies on some very special properties of the example. Suppose we set out to designing a robot to pull off bank heists, in contrast, and represent (in FKRL) the commonsensical fact that anything to the *right* of the robot is neither to the *left* of it nor *straight in front*. In order to motivate an appropriate M_c, we need to understand the relation between FKRL programs (now viewed as specifications) and FKRL impressions. So imagine the notation for FKRL programs is reminiscent of logical notation, and that we could "write down" something like the following "in FKRL"—which is to say, we could write the following FKRL expression *E* to serve as the external notation for the desired FKRL impression:

;*x* [RIGHT(*x*) ⇒ (¬LEFT(*x*) ∧ ¬FRONT(x))]

Suppose, furthermore, that this FKRL expression is more specific than the impression that it will generate in two ways. First, there is to be no fact of the matter, in the resulting impression, about what particular variable was used in the program; the expression might equally well have used *y* or *z*. Second, although matters of lexical notation force one of the conjuncts to be first (¬LEFT(*x*) in this case), we will assume that impressions are internally realised as unordered sets. Thus the following expression would have generated an indistinguishable impression:

;*x* [RIGHT(*w*) ⇒ (¬FRONT(*w*) ∧ ¬LEFT(w))]

[xx]Given these assumptions, we can then take on the task of providing a semantical analysis of FKRL programs—which is to say, an analysis of the relation between the FKRL expressive specifications and the resulting FKRL impressions—using the model-theoretic approach of indirect classification. It is unlikely that we would do no more than constrain the models of this impression to those that satisfy the logical implication, since we can presume that more fine-grained details of the impression's structure will play a functional role in licensing inference (such as the fact that the negation signs have not been pulled to the front, as they have in the semantically equivalent ¬'*x* [RIGHT(*w*) ∧ (FRONT(*w*) ■■ LEFT(w))]). So we might classify it using something like a term model, with the set of all equivalent expressions (including all those expressively differing only in the names bound variables and/or the ordered of conjuncts). Or if we were warranted in taking a more abstract approach, we might develop our analysis in terms of an interpretation function that mapped RIGHT, FRONT, and LEFT onto three distinct unary predicates, and classified the impression in terms of the set of all models satisfying the given implication.

To relate this to figure 8, I will use *E* for the quantified expres-

[x] «Check this paragraph for correctness and intelligibility ... seems awkward, and a bit incoherent re specifications and expressions?»

sion, *I* for the specified impression, C_1 for the first classification, C_1 for the second, and *W* for the impression's interpretation—as suggested in figure 9.[25] It should be obvious, first, that C_1 and C_2 are both more abstract than *E*, in the information-theoretic sense of being less rich. Second, C_2 is in turn more abstract than C_2, since this model makes fewer distinctions (identifying all semantically equivalent expressions). Finally, both C_1 and C_2 have additional properties that are, as we might say, "semantically inert": properties that in this application neither themselves are, nor do they model, nor do they classify any properties of *I*, *E*, or *W* (for example C_1 and C_2 are both sets, even though none of *E*, *I*, nor *W* is a set).

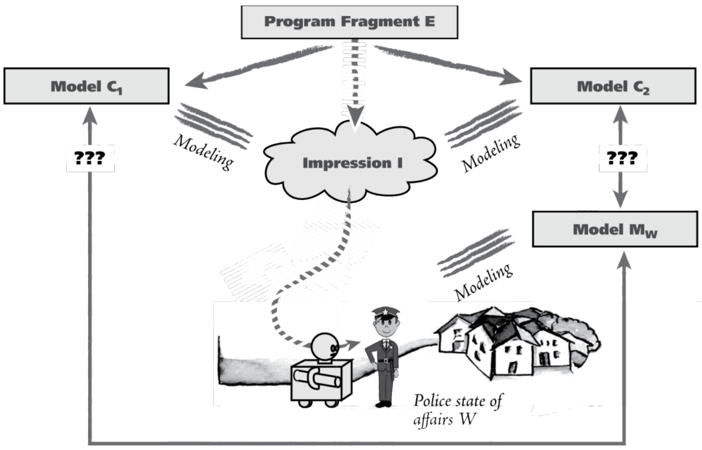


Figure 9: Indirect semantic classification of FKRL programs

Given all of this, we are finally ready to ask the question to which this has been leading: is either of C_1 or C_2 a candidate for being a model of W—i.e., a candidate for serving as a model-theoretic stand-in for the representational import of *I*? And the answer, to bring it all home, is *no*.

The fundamental problem is that "being to the right of" is not a one-place relation: one thing is "to the right" of another thing, in the world, *only relative to the position and orientation of the first*. Thus C_2 will not do, as a model of the representational content of *I*, since it does not contain enough information to determine, for example, whether impression *I* is *true*. If we wanted to model *W*,

[25] The impression is depicted as inside the robot's head because the real interpretation function is being understood as holding between the robot's mind and the policeman in front of it.)

then various additional circumstantial factors—including the position and orientation of the robot—would have to be brought in explicitly. In dealing with *W* we need to deal with actual position in the world, to put this another way ("in the world" is where ones encounters police).

There is no formal problem with adding circumstantial parameters to an interpretation function, and thereby distinguishing meaning and content. We saw how to do that in section 3. Rather, the point of the exercise is to see what it is that these circumstances affect: *the semantic relation between process (I) and world (W)*, in particular, *not the relation between program (E) and process (I)*.

[xx]It is not accidental that we are considering a context-dependent case, since context dependence (a virtually ubiquitous semantical phenomenon, in my view) brings into focus the absolute importance of locating all relevant semantical phenomena and relations in their proper place. [xx]It is far more likely that the machine's behaviour will revolve around regularities framed in terms of what's in front of it, to its right, or to its left, not in terms of what is in a given position. If the robot's external circumstances were mistakenly introduced in the $E \Rightarrow C_2$ relation, the resulting C_2 would fail as a model of *I*. For example, it would be of no help in explaining matters if *I* somehow broke and caused the robot always to ignore things on its left, since "on its left" would not be a notion in this modified C_2.

In general, of course, nothing prohibits a theorist's classifying something by its content (as we did in the factorial case). Exactly such a strategy, in fact, is arguably what underlies our standard (indeed, at the moment, only) way of describing the propositional attitudes constitutive of folk psychology (*'knows that'*, *'believes that'*, *'hopes that'*, *'fears that'*, etc.).[26] The point is only that we

[x] «This paragraph may need complete rewriting (certainly it needs to be thought through, carefully, in order to determine whether that is so) …»

[xx] «Is this sentence coherent?»

[26] Folk psychology faces exactly the same problem we have just surveyed. In particular: (i) it classifies people's mental states by content; (ii) the purpose of these classifications is to explain how people behave and what they do; and (iii) the content of people's mental states is determined in part by their

must not assume that all indirect classification is of this type. More seriously, simple indirect classification by semantical content will in general fail as a strategy for semantically analysing the impressions of circumstantially dependent agents.

6 Impressions, Expressions, and Complications

I said in section 4 that no there is no generally agreed, direct way of describing impressions. So far we have seen two quite different alternatives: a metaphorical approach, using the language of linguistic expressions (section 4), and an indirect approach, classifying them in terms of abstract mathematical structures (section 5). Before leaving the subject, we must recognise a third.

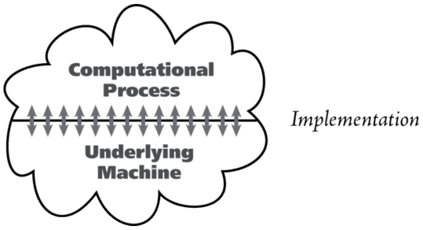


Figure 10: The Implementation Relation

It is common in informal AI practice, and standard in what is called 'operational' semantics in the programming language community, to describe the impressions and behaviour of a given computational process in terms of *the corresponding impressions or behaviour of a lower-level machine on which the process is implemented*. This relation is depicted in figure 10. For example, if we were to adopt this approach to analyse the semantics of FKRL impressions we might do so by presenting the Lisp code that has been developed to serve as the implementation of FKRL impressions.

From a theoretical point of view this approach is hardly satisfying, since it just causes the problem to recur at a lower level—raising questions about how to describe the implementing machine. In practice, however, it is widely accepted because it is often possible either (i) to refer either to a familiar underlying machine,[27] or (ii) to model the input/output behaviour of the result-

circumstances. These facts have led some writers, such as Stich (1985) to conclude that folk psychology will never be scientifically reconstructable, but in my view this seems to be an unwarranted pessimism; the problem, rather, is to see how folk psychology compensates for the external circumstantial dependence.

[27] As usual, and as the example about Lisp code suggests, practice is in fact one level more complex than this analysis suggests. One gives the opera-

ing machine in terms of ordinary mathematical functions. The relation between traditional denotational and operational semantics of programming languages, therefore, is primarily one of *abstraction*: by using coarse-grained functions as classificatory devices, the so-called "denotational" account gets at less detail than does the operational account. But the fact that they are theoretically distinct ways of getting at the same phenomenon is betrayed by the fact that it is standard practice to *prove the two types of account equivalent*. In particular, they are two different theoretical approaches to *analysing the nature of the computational process itself*; neither takes up the question to which we have been addressed: not of analysing the computational process qua process, as it were, but of analysing that process's semantic import![28]

For our purposes, the importance of this third approach lies in its introduction of implementation as *yet another intentional relation for semantical analysis to contend with*. As with representation

tional semantics of a programming language *L*, viewed specificationally, by translating expressions types of *L* into complex expressions types of programs, written in an implementing language *L'* that implements *L*. The language-process relation for *L'* is what is usually assumed.

[28] There was some misunderstanding, when 3-Lisp was introduced (Smith (1982, 1984)) about the two semantical factors in terms of which it was analysed and designed ('f' and 'c'>, they were called, but they corresponded directly to first and second factors in the framework being presented here). Unfamiliar with the two-factor framework, many computer scientists assumed they were merely new names for operational and denotational accounts, respectively. This was false, but in retrospect the confusion can be attributed to three things: (i) the fact that 3-Lisp was designed on an "ingredient" view of programs, whereas, as described in the text, programming language analysis is typically carried on within the specificational tradition; (ii) 3-Lisp's represented "world" was constrained to being one of pure mathematical abstractions and internal structures (since it was presented as a computational model of introspection), so that the *domain* that 3-Lisp impressions represented was the same one that would normally be used for both operational and denotational semantics—i.e., the domain of impressions and of the obvious mathematical models of them; and (iii) because of this restricted domain, the interpretations of 3-Lisp impressions were not dependent on external circumstances, so that the clear difference between model and interpretation, noted at the end of section 5, did not apply.

These three reasons conspired together; it has only been in the last few years that the various intricacies of their relationship have been clarified.

and belief, implementation is a directed, asymmetric, intentional notion: to say of *X* that it is an implementation is to imply the existence of a *Y* such that *X* is an implementation *of Y*. Furthermore, the implementation boundary is opaque to other semantical relations—i.e., it cannot be viewed as invisible modelling, or easily composed. For example, if we implement FKRL impressions in Prolog, and if the representational import of Prolog impressions can truthfully be given as standard first order model-theoretic semantics,[29] then it would not follow that the representational import of FKRL was the representational import of Prolog. At best the *interpretation* of Prolog impressions—the elements of Prolog's semantic domain—would be *FKRL impressions themselves*.

It is almost time to summarise the various distinctions we have made, and assemble a coherent overall picture. Before doing that, however, we must tie up two loose ends.

First, in the previous section we distinguished the representational content of impressions from the entities that theorists use to classify them indirectly, identifying a *modelling* relation between the two. But we have not yet taken this observation to its obvious conclusion: modelling, like representation, specification, knowledge, implementation, etc., is itself a semantic, intentional, notion. Like many other things we have seen, a model is not a

[29] I doubt this, for reasons that can easily be explained using terminology we have already introduced. As classically understood, standard first order logic is both declarative and syntactic, in the sense of section 2. Real-life Prolog programs, however, violate the assumed independence of factors: their role affects their import. Lacking techniques for spelling this out (ie., techniques for providing explicit two-factor analyses), most computer scientists who give semantics for Prolog programs in fact provide model-theoretic analyses of functional role, using term models and such, in the sense explained in section 5. Logicians, expecting analyses of representational import, quite reasonably find these reconstructions odd. Furthermore, to the extent that it is functional role, not representational import, that is retained, Prolog's claim to clear *semantics* is thereby undermined.

Note that a model-theoretic analysis of functional role (first factor), on the ingredient view of programs, is liable at least partially to coincide with a mathematical model of representational content (second factor) of the programs used (on the specificational model) to describe them. The subject matter is rife with such potential semantical confusions.

model all on its own; models are models *of* something. A balsa airplane, for example, might be a model of a real airplane no longer around, or of one being designed. Similarly, the sets of quadruples we have talked of are models of a Turing machine; the numbers 0 and 1 are often used as models of Truth and Falsity. Thus we need, ideally, to give a semantical analysis of the modelling relation, if techniques of modelling or indirect classification are ever used. I.e., in the terms of figure 9, we need semantic analyses of the $C_1 \Rightarrow I$ (or $C_2 \Rightarrow I$) and $M_w \Rightarrow W$ relations, as well as of $E \Rightarrow I$ and $I \Rightarrow W$.

Second, all the computational processes we have looked at so far are limited in the following obvious way: we have imagined them acting in the world (driving around, computing factorial), but we have not provided them with any *communicative* abilities. They cannot talk. In order to be realistic, therefore, we should complicate our pictures yet one more time, as indicated in figure 11. In order to contain the complexity, I have omitted all models and indirect classification from the diagram, showing only the genuine intentional relations that actually obtain in a given case. I will use the general term **notation** for the relation between expressions and impressions that they give rise to or express, and the more specific **internalisation** and **externalisation** to get at each direction of information flow. The analog, in the human case, is the relation between the sentences we speak and hear, and the impressions in our minds (mentalese or whatever) to which they correspond. To the extent that impressions are viewed linguistically, internalisation might be analysed as a species of translation, but it is important not to bias terminology in advance.

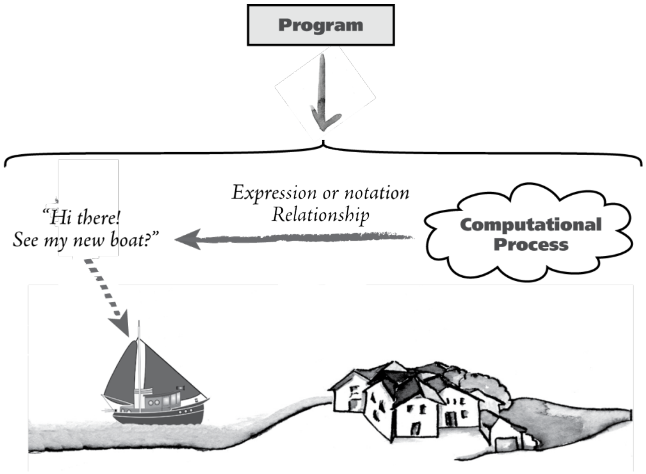

Program
Expression or notation Relationship
"Hi there! See my new boat?"
Computational Process

Figure 11: Programs that specify communicating agents

Issues of notation tie back to an issue we left unresolved above. Very often, the languages computer systems "speak"—query languages for data bases, editing commands for word processors, manipulation protocols for spread sheets—are visibly distinct from the programming languages used to create them. Many AI programming languages, however, such as Lisp, Smalltalk, Logo, and recent versions of Prolog, are primarily interactive, suggesting the third model of programming suggested in figure 6 (c), above. Furthermore, the increasing incidence of "user-friendly" computers suggests that this interactive model of computer language will only spread In addition, since it is the correct model for natural language, people will be biased towards an interactive stance to the extent that people understand computer languages by analogy to their native linguistic skills. Thus we have a genuinely triple ambiguity in the term 'program,' which only raises the chances of semantic confusion. Ironically, confusion between the specificational and interactive models of programming, coupled with the fact that the program⇒process relation is mediated by what is called an *interpreter*, has lead many computationalists to think of internalisation as the fundamental semantic relation—thereby embracing exactly the view that Lewis deridingly calls "markerese semantics."[30] On the other hand, AI practice suggests what Lewis's analysis does not: that internalisation is a substantial intentional relation in its own right. If nothing else, more adequate vocabulary might facilitate better interdisciplinary communication.

We are ready, then, to summarise four major themes in the investigation so far.

1. We distinguished *functional role* and *representational import*, and set ourselves the long-range goal of an integrated account of full significance, consisting of partially independent but coordinated accounts of each semantical factor.

2. We claimed that since we do not yet have adequate vocabulary for talking directly about impressions, we typically avail ourselves of any one of three alternative ap-

[30] Lewis (1972).

proaches:

a. Using *metaphorical terminology*, such as the language of linguistic expressions;

b. Using *indirect classification*, typically in terms of abstract mathematical structures; and

c. *Abstracting over implementations*, which makes the problem recur.

Differences among these alternatives, and differences in the fields in which they are popular, have obscured our ability to agree on underlying impression structure itself.

3. Setting aside considerations of functional role, we identified the following important relations, each at least a candidate for its own semantic analysis:

a. The *specification* relation, between a program and the process or impressions it engenders;

b. *Internalisation* and *externalisation* relations,[xx] between expressions used by a system to communicate with its users, and the impressions they give rise to or express;

c. The *implementation* relation, between impressions at one level of description, and other lower-level impressions in terms of which they are implemented; and

d. The *primary representation* relation, between impressions (process) and the states of affairs in the world with which the agent is concerned.

All four of these can be called *genuine*, in the sense that they are all a necessary part of the life of the representational agent in question—they have not been posited solely for purposes of theoretical analysis. Other relations between the same structures could be added, of which the most important is probably the relation between communicative expressions (language) and the world—the subject, in the human case, of natural language semantics. I

[x] «Check: I think I am confusing about the words 'notation,' 'externalisation,' 'internalisation,' etc. — including in the diagrams. Make it all consistent.»

will adopt these four relations, however, as primary, because they are all candidates for full two-factor accounts. Put another way, they are all of a *causal* nature, in a way that the direct relation between language and the world is not. Note also that impressions participate in all four relations (which puts extra pressure on our ability to describe them in their own right), being the semantic domain in the first three, the so-called "syntactic" domain only in the last.

4. In addition to identifying these genuine semantical relations, we uncovered numerous relations of *modelling* or *indirect classification*, cross-cutting all of the above three. To distinguish them from the genuine relations, I will call them **theoretic**, since they are introduced for the purposes of us, *qua* theorists, rather than for the agent itself. Nonetheless, if we as theorists employ them, they too must be semantically understood. If we were to use model-theoretic techniques to understand the four genuine relations listed above, we would bring to eight the total number of interacting correspondence relations. The complexity can get a little daunting. It is no wonder that it is sometimes hard to tell, when presented with a "semantic analysis," just what it means.

All these results contribute to the general series of challenges I am mounting against straightforward model-theoretic semantics.

1. The first specific challenge was implicit in our two-factor analysis itself, and its concomitant rejection of the independence of functional role and representational content.
2. The second arose when we removed the constriction that impressions be syntactic or linguistic in nature, and embraced instead a much wider range of representational possibilities.
3. The third challenge stems from the multitude of genuine intentional relations just cited—*specification, internalisation, implementation, representation*, etc.—more than one of which will require its own two-factor analysis.
4. The fourth derives from the fact that standard theoretical techniques of indirect classification and modelling intro-

> duce, at the level of theory, a whole spectrum of additional correspondence relations, at least distractingly similar to semantic relations, if not semantic relations in their own right. If we do not understand them they will pollute our attempts to clarify the semantic relations we are primarily interested in.

Nor are we done raising challenges. In the next section I will turn to a fifth, coming to a sixth at the end of the paper.

7 The Correspondence Continuum

I said in section 3 that the model-theoretic tradition characteristically assumes a non-transitive denotation relation, motivated by clear linguistic cases: an English description of a French description of dessert, for example—such as "the four words *neige, la, a*, and *oeuf*, in reverse order"—is a description of *language*, not a description of something to eat. At the same time, we saw traditional analyses freely compose modelling relations, as for example when a number encoding a description of a Turing machine is identified with the Turing machine in question. This free composition goes hand in hand with modeling's traditional invisibility.

Unfortunately, however, these two cases—non-transitive denotation, and transitive modelling—do not cover the whole spectrum of semantic relations. In the general case, intentional relations combine in much more complex ways. We will look at three examples.

First, suppose I remark on a photograph you have taken of one of my favourite sailing ships, and you then present me with a copy made by photographing the original. It would be pedantic for me to maintain, on grounds of use/mention hygiene, that the copy is not a photo of the ship, but rather a photo of a photo of a ship. For most purposes, the relation between the copy and the original print is sufficiently close that I can harmlessly compose the two correspondence relations (copy-original and original-ship), yielding a result (copy-ship) essentially identical to the second. But not for all purposes: if, on close inspection, I claim that there is a tear in the ship's sails, you might appropriately reply that *no*, the tear, rather, is in the original photograph that the copy was made from. Or I might be interested in the quality of your photographic technique, and use the copy as a representation of your original work. The appropriateness of the ability to compose, or to "look through" a copy to what is represented, can depend on the purpose to which a semantic relation is put.

Second, imagine connecting an FKRL system to a visual recognition system, consisting of a TV camera, special-purpose line-finding hardware, a figure-recognition module, etc. In such a case one might be tempted to say that the configuration of pulses on the cameras represented in the intensity of incoming light, and that the resulting FKRL impression represented the object under view. Yet although the former objects play a causal role in supporting the latter, it is not clear how the two representation relations fit together—the second seems to "leap completely over" the first. In spite of systematic correspondences among the constituent structures, the representation relations seem curiously independent. It is as if the structural correspondences compose, but the representation relations do not.[xx]

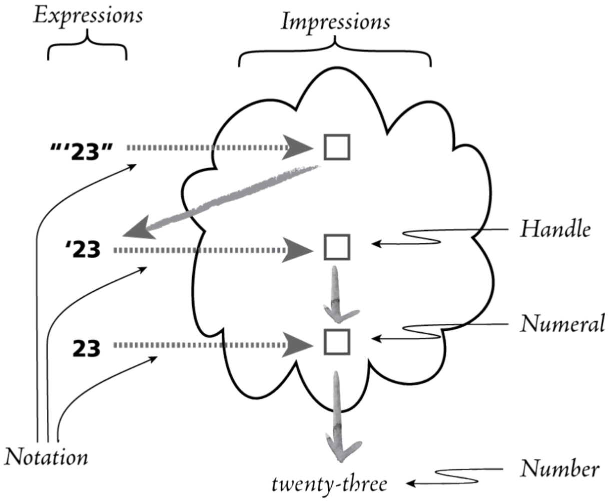


Figure 12: 3-Lisp's plethora of representation relations

Third, in designing 3-Lisp, I distinguished impressions called *numerals* from canonical impressions denoting them (identified as a species of *handles*), in spite of the fact that the denotation relation was an exact isomorphism. I did so because, trained in avoiding use/mention confusions, and viewing impressions as analogous to language, I thought representation relations *could not* compose. Various colleagues suggested that this strictness bordered on pedantry, and recommended that I simply identify the two impressions. Others even suggested that I identify both of them with the number designated, since as far as they could see

[xx] «That paragraph may need substantial help … »

the impression-number relation was also one of isomorphism.[31] But my allegiance to semantic strictness was strong: as shown in figure 12, I refused to say that the two-character expression written ‘23’ (without the quote marks) represented the number twenty-three; rather, when speaking carefully, I said that it *notated an impression that designated that number*. Similarly, I was forced to say that the three-character expression ‘ '23 ‘ (i.e., a single quote mark prefacing the two-digit numeral) notated a handle impression that designated a numeral impression that designated a number. By the same token, the five-character expression ‘ “'23” ’ notated a handle that designated an *expression* that notated the numeral impression that designated the number. And so on.

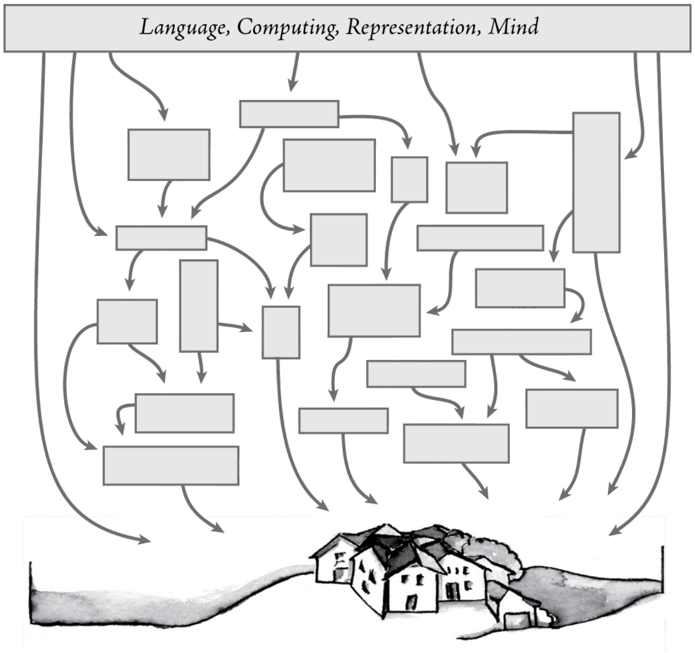


Figure 13: Semantic Soup: The Correspondence Continuum

While 3-Lisp was certainly semantically clean, in retrospect some of its rigidity seems gratuitous, even if I remain opposed to any *identification* of strings with impressions, or of impressions with numbers. It is overwhelmingly convenient to be able to point to a figure on a computer screen and say, simply, that it represents a number. More seriously, it is not obvious that one might not even be *correct* in doing so. And yet at the same time there are occasions when it is crucial to distinguish among expressions, impressions, and numbers.

All of these examples illustrate my fifth challenge to traditional model theory: *neither*

[31] In point of fact only one factor of the full significance was an isomorphism.

strict non-transitivity, nor indiscriminate identification, is always appropriate.[x] In each cited case, as so often happens, theoretical technique is not up to the demands of practice. The true situation is more accurately pictured in figure 13. The idea is this: a given intentional structure—language, process, impression, model—is set in correspondence with one or more other structures, each of which is in turn set in correspondence with still others, at some point reaching (we hope) the states of affairs in the world that the original structures were genuinely about.

It is this structure that I call the **correspondence continuum**—a "semantic soup" in which to locate transitive and non-transitive linguistic relations, relations of modelling and encoding, implementation and realisation, the rest. Several points are important.

First, I will not presume, in the general case, anything about composition, relative structure, circumstantial dependence, or any other traditional issue: such questions will have to be answered individually, based on particular facts about specific cases. Sometimes, and for some purposes, these representation relations will happily compose; other times not. Sometimes *some* properties (such as ambiguity!) will be preserved even across a whole string of such correspondence relations, even though other properties (such as one-to-one correspondence of objects) are lost. In the next section I will begin to sketch out an analysis of correspondence relations that will show how this might go.

Second, one should not think of this as necessarily a single dimension; the diagram is meant to be able to accommodate the multiple dimensions of representation (notation, representation, specification, etc.). As we have just seen in 3-Lisp's case, and as we saw so often in the last section, part of the task, in analysing the semantics of computational processes, is to tie together different correspondence relations that are neither totally independent, nor arranged in a simple linear order.

The general picture given in figure 13 is intended as a replace-

[x] «Say, somewhere—perhaps here, or anyway point to it here, even if the main point is made elsewhere—that these are the sorts of thought that have led to the design of the fan calculus.»

ment for the simplistic diagram of figure 2, even for the most basic intentional relations. In the remainder of the paper I will try to address a few of the numerous questions it raises.

Here is one, for starters. Which, if any, of these correspondence relations should be counted as genuinely semantic, intentional, representational? Surely not all. For example, to take another visual example, at the very moment I write this there is a series of correspondences of some sort between activity in my visual cortex, the signal on my optic nerve, the pattern of intensity on my retina, the structure of the light waves entering my eye, the surface shape on which the sunlight falls, and the cat sitting near me on the window-seat. And yet it is the *cat* that I see, not any of these intermediary structures. A causal analysis of perception, that is, would require a cascade of correspondences, but in this case only the full composition, *but not any of the ingredients*, would count as a genuine representation (though it does not follow that these intervening structures are thereby any less important). Similarly, even if I indirectly classify impressions with functions from possible worlds to states of affairs, and then map those mathematical structures onto genuine situations in the world, the agent itself will attend only to the situations in question, entirely unaffected my abstract classifying structures.

Both of these cases, and many of the phenomena cited in the previous section, suggest that the number of important correspondence relations greatly outstrips the number that are of a genuinely semantic or intentional nature. Such arguments lead to a simple and obvious conclusion: *critical correspondence of non-identicals is a far more general phenomenon than representation or interpretation*.[32] First, it permeates theory, in terms of indirect classification and modelling. Second, it permeates practice, as manifested in such notions as implementation, encoding, realisation, presentation, specification, internalisation, and externalisation, as well in as our initial concerns of representation and knowledge. Third, although not all these correspondence relations should be counted as fully intentional, there is no chance

[32] This implies, of course, that there must be much more to representation than correspondence. Hence footnote 1 «check»; correspondence on its own requires neither disconnection nor registration.

that we will understand semantics unless we are first clear on how they all fit together. So my recommendation is that we peel correspondence away from more difficult semantic issues, and make it a subject matter in its own right.[33]

Let us look, then, at what a theory of correspondence might be like, before returning to semantics and to knowledge representation.

8 A Sketch of a Theory of Correspondence

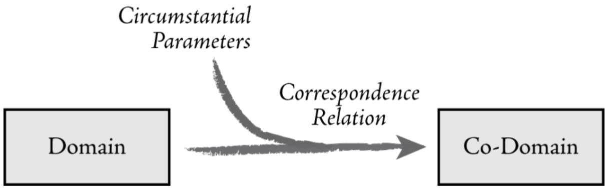


Figure 14: The General Structure of Correspondence

In broad outline, I will adopt a quite simple approach to the structure of correspondence. First, I will identify two domains, presumed to consist of a pre-determined collection of situations, objects, properties, and relations. Call them **domain** and **co-domain** (though this is not category theory), and say that an element of the domain **cor-**

[33] Strictly speaking I do not believe this, for two reasons. First, my metaphysical predilection is to attribute the notions of object, property, and relation to a collaborative interaction between mind and world, so that the world *alone* need not be held responsible for objects' boundaries and kinds (naive realism), nor need they be viewed as pure constructs of cognition (variants of solipsism or idealism). Second, I am at least prepared seriously to entertain the hypothesis that minds, fundamentally, are embodied representational processes. In conjunction these two views raise the following "chicken and egg" problem: if minds are required in order to know how the world is structured, and if minds are representational, then representation must seemingly be studied before correspondence, in order to establish the categories in terms of which the correspondences will be articulated. On the other hand, for reasons spelled out in the text, I think the chances of getting representation right without a prior theory of correspondence are rather limited.

These considerations interact with another distinction. Which person is being held responsible for the categorisation of the domains in question: the agent under study, or the theorist? I assess the interaction among these issues in Smith ⟦forthcoming (b)⟧; the net result is simply the rather predictable conclusion that the two notions (correspondence and representation) must be viewed as something of an indissoluble pair. This conclusion, however, does not in any way challenge the view being expressed here: that they are not the *same*.

responds to an element of the co-domain. Furthermore, without introducing any assumption of symmetry, I will speak most generally of correspondence *relations*, rather than functions, and make room for circumstantial parameterization in the usual way. The situation is pictured in figure 14. (The resemblance to figures 2 and 3 is obvious; we can now see those figures were right for correspondence, but wrong—because too simple—for the complex general story about semantics).

Given these two domains, specific correspondence relations are defined between states of affairs in each domain—not between the domains themselves, nor between objects, properties, or situations on their own, but between *things being a certain way* in one domain, and *things being a certain way* in the other. Thus, the light's being red corresponds (or so we hope) to cars' stopping. Similarly, we might say that the sequential concatenation of the numeral '2', the sign '+', and the numeral '3' corresponds to the addition function's being applied to the numbers two and three, which in turn corresponds to the number five.[34] Even in cases where there is a simple correspondence of objects, as when the numeral '3' stands for the number three, it is really the object's *being that and not some other numeral* that corresponds to the number's *being that and not some other number*. The numeral may have all sorts of other properties—such as consisting of one curved and one straight line—which do not correspond to anything in the co-domain at all.

There are several reasons to require an explicit specification of domains, and to lay responsibility for the correspondence relation on states of affairs (rather than on objects *per se*). In general, objects exemplify infinitely many properties, and participate in infinitely many relations—in this sense the world is overwhelmingly rich. Even questions of object identity do not escape this richness, as precise attempts to define numerals quickly reveal (does the expression "124+124" contain one, two, three, or six numerals?). It is therefore necessary, in characterising a particular correspondence relation, to identify in advance the particular set of objects,

[34] Note that this phrasing suggests iterated correspondence: expressions to function applications, and from there to values. The connection between iterated correspondence and so-called "intensional" analyses of functions and relations is discussed at the end of this section.

properties, and relations in each domain that are constitutive of the significant states of affairs—what I will call a prior **registration**[35] of the domains—and then to identify, with reference to that registration, how states of affairs in the domain correspond to states of affairs in the co-domain. This is partly because states of affairs, at least as I am using the notion,[36] are individuated by the relations and properties they instantiate (a number's being the sum of two plus two, and the same number's being the positive square root of sixteen, are different states of affairs, on this view). But it also seems true to common sense, as the red light example suggests.[37]

(As well as adopting these two theoretical assumptions, there is another which I will explicitly set aside. Many writers, including theorists as far back as Peirce, have expressed the deep intuition that representation is a three-place, not a two-place, relation, involving not only representation and represented, but also *interpreter*, *observer*, or, in Peirce's case, *interpretant*. Thus a text, and probably even a simple map, is taken not to be a representation on its own, but to represent only for some other agent or purpose (or both). I sympathise in the representational case, but we are talking here about a simpler notion of correspondence, where the question is much less clear. For example, one could view a binary correspondence relation between *X* and *Y* as a relation that an interpreter posits or reacts to, in taking *X* to represent *Y*. Thus your map may not represent New York unless you or some other person takes it to do so, but that act of taking it to represent New York involves attributing or establishing a *binary* correspondence relation of a certain type—of a type, furthermore, that might be characterised in terms of the theory I am proposing. In addition, given my general recognition of the importance of circumstantial

[35] «Point towards O3 and registration; is this the first place I use the term?»

[36] My intention is to employ the term in a way compatible with its technical use in Situation Theory [Barwise, 1986a], although nothing in the text requires that particular analysis.

[37] The theoretical stance of taking registration as prior to correspondence, and correspondence as at least partially independent from representation, is not one I am ultimately satisfied with; see footnote 18 «check», and Smith ⟦forthcoming (b)⟧. It seems well motivated, though, at least as a way of getting to the next stage in semantical clarity.

dependence, it is not obvious that the role of interpreter has been excluded. But however this goes—and even if one were to argue convincingly that even correspondence should be analysed as tripartite—my present purpose is to define a project, not to report on its conclusion. Such questions should ultimately be answered by theory, not prejudged. And I would hazard that the distinctions to be made, here, in terms of correspondence treated as binary would carry over, though perhaps be thereby complicated, in a three-element version.)[38]

I will call the relevant states of affairs in the domain and codomain the **source** and **target**, respectively. So the source expression "72°10′ E, 44°20′ N" might correspond to a bucolic target in northeast Vermont. In general, correspondence relations will be defined in terms of source and target **types**, in such a way that instances of the source type would correspond to instances of the target type in some determinate fashion. For example, the mapping from sets of quadruples to Turing machines would be established so that a *particular* quadruple's having certain elements would correspond to the controller of the corresponding Turing machine's satisfying a *particular* transition function (though *what* Turing machine that transition function was a transition function *of* might be assumed, for the whole set of quadruples, and thus not explicated "corresponded to" by anything). This approach makes sense of the intuition about modelling suggested in section 5: that what is *specific* (or particular) about one state of affairs—the source—determines what is *specific* (or particular) about another—the target.

In setting out an initial analysis of this sort,[39] I call a particular correspondence relation **iconic** if each object, property, and relation in the source corresponds, respectively, to some object, property, and relation in the target. I.e., the abstract type (object,

[38] In cases where a third agent—an interpreter—is present, a possible solution is presented to the problems raised in footnotes 18 «check» and 21 «check»: the agent can' register both representation and represented. But there are two problems with this. First, of course, we have to ask how agents register, which brings the problem back to roost. Second, it is a strong and possibly false claim that interpreters register signs and language they use (as opposed to mention).

[39] «See Smith (forthcoming(c))—check!»

property, or relation) of the source is the same as the abstract type of its target. A particularly important case of iconicity occurs when a source object, property, or relation corresponds to *itself* in the target: I will say in such a case that the target structure is **absorbed** in the source. For example, left-to-right adjacency is absorbed in the grammar rule "EXP ⇒ OP(EXP,EXP)" for a simple term language for arithmetic. Similarly, to suppose that the necessity of set membership, in a model-theoretic analysis of modality, models necessity in the world is to assume, counter-factually, that necessity is absorbed. In contrast, a target property or relation is said to be **reified** if it is corresponded to by an object in the source (reification is not defined on objects). Thus for example the syntax of predicate calculus reifies properties, because it represents them with (instances of) predicate letters, which at least in standard syntactical analyses are registered as objects.

A correspondence relation is called **polar** when an existentially positive source (something's being the case) corresponds to an existentially negative target (something's not being the case), or vice versa. Hotel lobbies provide an example, where a key's being present in the mail slot at the registration desk indicates the fact that the client is gone. A relation is called **typological** if it can be defined without reference to distinguished individual objects in the domain or co-domain. Thus the standard Cartesian relation of ordered pairs of real numbers to points on a plane fails to be typological on four counts: *origin*, *orientation of x-axis*, *unit length*, and something to distinguish *left and right orientation*, such as a distinguished normal to the plane. Finally, when either or both domains are analysed mereologically—in terms of notions of part and whole—either or both ends of the correspondence can be defined **compositionally**, in the sense that what corresponds to (or is corresponded to by) a whole is systematically constituted out of what corresponds to (or, again, is corresponded to by) its parts. If the part/whole relation is itself absorbed, a very strong version of compositional correspondence obtains, where parts of a source correspond to parts of that source's target.

Many other such relations can be defined, ranging from this simple sort up through more complex cases having to do with sentences, quantification, use, circumstantial dependence, etc. The intent here is not solely to develop a theoretical typology (though that is often useful, especially early in theoretical development), but eventually to identify an algebraic basis of correspondence in terms of which to analyse arbitrary relations. Given such an algebra, for example, and an analysis of two relations C_1 and C_2 in terms of the orthogonal set of basic features, it should be possible to predict the exact structure of the composed relation $C_1 \circ C_2$. Thus we would expect the composition of two iconic relations to be iconic, iconic relations to be both left and right identities (with respect to this algebra), and so on and so forth. Note, however, that the appropriateness conditions for composition are very strong: $C_1 \circ C_2$ makes sense only if the targets of C_1 are of *exactly the same type* as the sources of C_2. Traditional isomorphism will not do, since isomorphism is just another correspondence relation C_3; the combination would have to be analysed as $C_1 \circ C_3 \circ C_2$.

As the isomorphism example suggests, a correspondence theory of this sort would provide theorists (I primarily have semanticists and computer scientists in mind, but of course the account would be general) with an extraordinarily fine-grained pair of glasses with which to analyse arbitrary structured relationship between domains. Every conceivable coding, representation, modelling, implementation, and isomorphism relation would be made blatantly visible. Whereas category theory can be viewed as highly abstract, in other words, correspondence theory would be exactly the opposite: *unremittingly concrete.*[x] This does not mean that abstract objects could not be studied within such a framework, of course; only that *no further abstraction by the theory* would be permitted unless explicitly accounted for (beyond that provided by the initial registration of the domains). Thus, whereas a model-theoretic analysis of the interpretation of the English word 'cat' might map it onto a mathematical set, a correspondence-theory based semantic account could not do so (or if it did, it would be wrong). There is no problem in providing a correspondence-theoretic analysis of the relation between the word 'cat' and the set-theoretic structure used by model theory to *classify* it, but that, as the correspondence theory would make explicit, is quite a

[x] «Say more about this divergence with category theory—and also point towards the criterion of concreteness in O3.»

different thing.

It is a consequence of this fine granularity that many standard mathematical techniques, such as that of identifying structures "up to isomorphism," would be inapplicable. But this result is to be expected: since the whole point is to avoid gratuitous modelling, and to explain arbitrarily fine-grained distinctions, the theory cannot indulge in any loss of detail.

As well as focusing on the detailed structure of specific correspondences between states of affairs, an adequate theory would have to address general questions about particular relations, such as whether every source in the domain corresponds to exactly one target, whether every target has a source corresponding to it, etc. It would be natural, that is, to define correspondence versions of such standard notions of totality, completeness, and ambiguity. But this starts to feel a little odd, because of its familiarity. Are we just reinventing traditional mathematical accounts of functions and relations? How do our categories of correspondence relate to such standard notions as isomorphism, homomorphism, injection?

The answer appears to be the following. It has often been pointed out that standard so-called *extensional* analyses of functions and relations, in terms of piece-wise pairings, ignore the structure of the connection between the domain and co-domain, even though that structure is often important in practice—such as when the function is to be *computed*, or the relation *recognised*, or when the connection is *causal*, defined in terms of the constituent properties. Extensional mathematical analyses abstract away from such concerns; when we describe functions in natural or formal languages, however, or embody them in machines, we typically betray a great deal of additional information. Thus the standard term designating the factorial function

```
if n =0 then 1 else n–factorial(n–l)
```

implicitly suggests a way of computing factorial, even though that information is lost in the standard extensional analysis, which would merely map the foregoing expression onto the infinite set of ordered pairs {<0,1>, <1,1>, <2,2>, <3,6>, ...}.

In the general case the information conveyed by a functional description can be sorted into three kinds, as suggested in figure

15: information about (i) the structure of the domain, (ii) the structure of the co-domain, and (iii) the structure of the relation between the two (the first two clearly merge when, as is often the case for simple functions, the domain and co-domain are the same).

Recognising the importance of this other information, various people have attempted to develop what are called *intensional* analyses of functions, relations, etc. The idea, or so it is claimed, is to make this extra information explicit. But from our point of view there is something curious about the way in which this is traditionally done. Because these efforts have arisen in the context of computation, recursive function theory, and a general concern with procedures, the approach is in fact not one of making these three kinds of information explicit, but rather of making explicit *the structure of an algorithm for computing the function* (or relation). Thus Moschovakis (1984) has proposed treating an algorithm as a first class mathematical entity in its own right, and a variety of writers have at least argued for dealing directly with procedures, such as those recommending procedural treatments of semantics.[40]

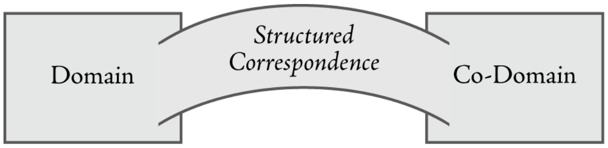


Figure 15: The three structures of correspondence

There is nothing wrong with explicating the notion of an algorithm, of course. But there is no reason to suppose that, even if successful, this project will make explicit the three kinds of information cited above. For example, no matter how explicit I am in giving you directions for driving across Boston, the structure of the city will at best be borne implicitly in the resulting descriptions of routes. Imagine trying to reconstruct a Boston city map by sorting through every route traveled by a long-time cab driver, gradually culling information about the town from such sequences as "Drive two blocks up Trapelo Rd, turn right on Grove," or heroic attempts explain how to get from Jamaica Plain to Logan airport without using a tunnel. Making the *algorithm*

[40] E.g., see Woods (1981).

specific will not even make explicit the structure of the relation it computes, let alone the structure of the related domains.

In contrast, a correspondence theory can be viewed as almost a dual project: it would provide an informationally rich account of the structure of the relation between structured domains, though it would remain silent (unless that project were explicitly taken up) on any question of *computing* this relation. It would get at the three relevant structures (of domain, co-domain, and correspondence) directly, rather than taking them to be indirectly manifested by specific ways of going from a given domain element to its corresponding co-domain element.

As for which project has a better claim on being an "intensional" analysis of functions and relations, I cannot say—nor, presumably, does it matter. For one thing, the very theory of correspondence I am proposing will among other things obviate the worth of such terms as "intensional" and "extensional." More important is to recognise the essential difference, and compatibility, between the two accounts. As suggested in figure 16, the distinction between fine-grained ("intensional") and coarse-grained ("extensional", or piece-wise) analyses is orthogonal to the question of effectiveness or computation. We can thus classify the standard set-theoretic model of functions and relations as *coarse-grained and non-effective*, recursive theory as *coarse-grained but effective*, and the theory of algorithms as *fine-grained and effective*. A theory of correspondence then occupies its rightful place as the fourth possibility: a *fine-grained but non-effective theory of relationship*.

	Effective	*Non-effective*
Fine-grained	Algorithms	**Correspondence**
Coarse-grained	Recursive Functions	Mathematical Functions & Relations

Figure 16: Analyses of Relationship

The location of a correspondence theory in this diagram is well suited to the semantic purposes for which we have needed it. One of the most fundamental facts about most genuine semantic relations, such as reference, is that *they are not computed*, in any coherent sense of that word. When I say "Bach died in 1750," and thereby refer to a long-dead composer, nothing *happens* in order to make the reference work; it just *is*. It is thus entirely to be expected that semantical examples should push us towards a fine-grained but non-computational analysis of structured correspondence.

9 Semantics Revisited

The availability of a correspondence theory would change semantical analysis in at least these ways:

1. As promised, the following traditional notions would be replaced: (i) a strict hierarchy of (meta-)languages, (ii) invisible but promiscuous modelling, and (iii)) the notion of an absolute use/mention distinction.

2. It would provide the theorist with sufficient equipment to analyse such otherwise unanalysed notions as *encoding*, and to discern and thereby avoid problems of gratuitous artifacts.[x]

3. It should provide, for the first time, adequate vocabulary in terms of which to analyse and assess such non-linguistic representational structures as images and analogue representations.

4. It would enables us to explain some lurking problems and unexplained worries that have plagued traditional approaches.

I will look at each of these briefly.

First, dismantling an absolute use/mention distinction does not mean licensing automatic composition of all correspondence relations. On the contrary, the intent of the algebraic basis of correspondence sketched in section 8 is to enable us to see what sorts of properties will propagate through iterated correspondences, and which ones will not. The popular closed-world assumption in AI, for example, is in essence an assumption that *object identity is absorbed*; in any given application it should be straightforward to verify whether this property is preserved across one or more correspondence relations in question. Similarly, the assumption that

[x] «I should include an example of what I mean by 'gratuitous artefacts'—presumably non-significant properties of models?»

words have referents could be justified, even by someone committed to the logical priority of mental impressions, just in case the internalisation and representation relations could be unproblematically combined. Even in written natural language, use vs. mention apparently shades off into matters of degree; thus we have (in something like increasing "semantic withdrawal"):[41,42,xx]

1. Margaritaville is lively;
2. Margaritaville is so-called for dubious reasons;
3. They call it *Margaritaville;*
4. When I asked where they lived, they said "Margaritaville";
5. "Margaritaville" is a fictional name
6. I am sorry to have to be the one to tell you, but "Marga-
   rita-
   ville" is hyphenated;
7. "Margu[illegible]ville" is smudged.

Particular analyses of use and mention would depend on the semantic relations employed; once again letting go of the strict theoretic distinction paves the way for accommodating a wealth of familiar facts.

As well as undermining use/mention distinctions, the correspondence continuum challenges the clear difference between "syntactic" and "semantic" analyses of representational formalisms—an especially important consequence given the allegiance commanded by this historically entrenched distinction. On the face of it, it might seem that we are simply removing an important method of discriminating accounts, which would be a negative re-

[41] Acceptance of the last two seems to vary, among people I have informally surveyed.

[42] Introspection suggests that quotation marks are primarily, if not always, used to refer to linguistic types. As a possible counter-example, Geoff Nunberg has suggested: " 'Fiat lux' started this whole mess' ", but at best that refers to an utterance of the Latin sentence different from the (enclosed) one used to refer to it. There does seem to be merit to the view that quoted expressions cannot be used to refer to their constituting internal *tokens*.

[xx] «Point out that the referent of 'Margaritaville' differs in all of these cases—yet to come up with a semantic analysis that identifies, in advance, all of the Δs that the full range of quotation requires would lead to untenable pedantry …»

sult. The claim, though, is that no *simple* "syntactic"/"semantic" distinction gets at a natural joints in the underlying subject matter, no matter how profound the ultimate difference, as it were, between map and territory.

For example, many writers have claimed to provide semantical analyses using models set-theoretically constructed out of basic syntactic elements such as sentences, ground terms, etc. (i.e., so-called "term models"). A typical AI case is found in Moore and Hendrix's proposal for a semantical model for belief;[43] similarly, term models are often used in giving semantical analyses of logic-based programming languages, such as in Goguen and Meseguer's EQLOG.[44] Although stamped with the official "semantics" insignia, they are often used as abstract models of (i.e., to classify) syntactic or computational properties, such as inter-reducibility of terms in a rewrite system (a-interconvertibility in the l-calculus, for example), effective derivability, etc.

My point is not to indict this practice, nor to dispute its theoretical importance. Rather, the point is this: *if* one is committed to a simple binary "syntactic"/"semantic" distinction, as on the traditional view, then such proposals would have to be counted as syntactic, and hence as false advertising—since for example the semantical interpretation of a formula such as DEAF(BEETHOVEN) would have only to do with syntax, nothing to do with the composer himself. On the more complex view we are proposing, needless to say, room is provided for such analyses as these. Whether they are labeled 'semantical' becomes a substantial issue—but the main point is that the theorist would need to make plain exactly what kinds of relations are being analysed, what kinds of facts or properties or states of affairs (e.g., in models) are being used to classify what others; what relations in the overall picture are computational, representational, whatever. The crucial points are just two: (i) the space of possibilities is not constricted in advance, by the nature of the theoretical framework; and (ii) a substantial (and presumably intellectually hygienic) premium would be put on stringent honesty about what is being claimed to be what.

[43] Moore and Hendrix (1979).

[44] Goguen and Meseguer's (1984).

The second main consequence of the new approach arises from its fine-grainedness, which thereby facilitates direct views onto otherwise invisible relations. These last fall into two kinds: (i) subject-matter relations that have heretofore evaded satisfactory analysis, like encoding and implementation; and (ii) theoretic relations like modelling, which have affected and sometimes distracted analysis. With respect to this fine-grainedness of approach, correspondence theory can be understood, in its relation to traditional semantics and model theory, as analogous to the relation between situation theory[45] and traditional set theory. In both cases, the classical system makes far fewer distinctions than at least some analyses demand. Thus situation theory, like other property theories, populates the world with properties, relations, facts, states of affairs, and the like, thereby embracing a much richer ontological foundation than the set theory we are used to. My brief against traditional model-theoretic analyses of languages and modelling is similar to Barwise and Perry's against set theory: it glosses much of the very detail we need to understand. Moreover, the enterprises of situation theory and correspondence theory are related in much stronger ways than by analogy. Any candidate correspondence theory will have to be based on a much richer ontological foundation than is espoused in set theory, for at least the following reason: in virtue of its explicit rejection of invisible modelling, correspondence theory will have to be able, in its own right, to cope directly with the full registrations of domain and co-domain.

For example, suppose someone wanted to use the proposed correspondence theory to assess the familiar representation relation between pairs of real numbers and points on a plane. In the model-theoretic tradition, the first job would to develop models of both phenomena. However, since ordered pairs are an eminently good model both of themselves and of points, the representation relation would look to be one of identity. For a correspondence theory to see the relation, it would have to license both ordered pairs of real numbers and points on a plane as legitimate, distinct, entities—as *first class citizens,* to use the computational phrase. Thus a set-theoretic base would simply not work.

[45] Barwise (1986a).

Given an adequate ontological foundation, however, and a concomitant account of correspondence, one should be able to repair some well-recognised lacks in current computer theorising, all of the "too coarse-grained" variety. The broad metric of Turing equivalence (relied on to demonstrate the "equivalence" of various models of computing) is a particularly blatant example—since virtually every imagined computer language, modulo standard idealizations of indefinite memory and time, turns out to be of equivalent power. The problem is that the *very notion of Turing equivalence itself* rests on promiscuous modelling; in showing one machine equivalent to another, one does not *really* show them to be the same; rather, what is shown is that one can *implement* one in the other. More seriously, all sorts of rather close correspondence relations—implementation, encoding, modelling, etc.—have similarly fallen between the cracks of theoretical assessment, being "closer," so to speak, than is typical of the representational import of language, but still distinct from identity. The hope is that a proper categorisation of correspondence will be a first step towards more adequate foundations and more subtle comparisons.

The third semantical consequence has to do with the potential integration and unified treatment of a wide variety of apparently disparate kinds of representation. Ever since the earliest days of Artificial Intelligence debates have raged about the relative merits and properties of so-called *analogue, pictorial,* and/or *imagistic* representations, vis. a vis those that are *sentential, propositional* or, as Sloman calls them, "Fregean."[46] Maps and diagrams are paradigmatic examples of the former; natural language sentences and formulae in first-order logic, of the latter. In spite of a diverse literature probing these distinctions and explicating cross-cutting distinctions buried in them,[47] however, no comprehensive framework has emerged in which to reconstruct the underlying insights. It is difficult not to notice that writers on these topics often refer back to Wittgenstein and Peirce, who wrestled with

[46] Sloman (1975).

[47] A representative series of articles by Dennett, Fodor, Kosslyn & Pomerantz, Pylyshyn, and Rey can be found in part two (*Imagery*) of Block's (1981). See also Sloman (Pylyshyn (1984); Sloman, (1975) and Pylyshyn (1984 chapters 7 & 8).

these issues before the development of modern semantical technique.

This literature conveys an unmistakable picture of complexity inherent even in the most paradigmatic examples. For example, Sloman (1975) attempts to differentiate *analogic* and *Fregean* representation by supposing that the former manifests a certain kind of correspondence (he neither explains nor constrains it) between the *part* structures of representation and represented. On the face of it, this would seem to amount to a structural correspondence between relations, of the sort we saw in discussing iconicity, coupled with a mereological registration of both source and target domains. The pure characterisation, in other words, seems exactly the sort that a correspondence theory should be able to explicate. Sloman's proposal, however, seems much less successful as a way of clearly discriminating between analogue and propositional representation. For example, as many have pointed out,[48] it does not have the intended bite unless one ties down the notion of "part." For a bar chart to remain analogue, the conception of part in the target domain must be taken quite liberally; on the other hand, such sentences as "Adrian, Amelia, and Aaron arrived in that order" seem to employ part relations in source (sentence) structure to signify part relations in the target (what is described). So the distinction is not so clear. Furthermore, there is no doubt that even paradigmatic analogue representations or images represent only with respect to a correspondence relation,[49] so the constraint on mereological correspondence would need to be spelled out, in exactly the way that the proposed algebra of correspondence types suggests.

Without delving into specific examples, several general things seem clear. For one thing, the persistent intuition that representations come in a wide variety of kinds seems exactly right. For another, analysing these kinds will require exactly the sort of fine-grained correspondence theory we are proposing. Finally, it is unlikely that common examples will sort into any small, mutually exclusive, set of nameable classes. Instead, we should license a full range of types of correspondence, kinds of circumstantial depend-

[48] See for example the discussion in Pylyshyn (1978).
[49] See for example Fodor (1975).

ence, and varieties of registration (continuous, discrete, compositional), in terms of which subsequently to characterise pictures, maps, graphs, schedules, models, images, and so forth, as well as sentences, formulae, and elements of language. The latter group, one would guess, will in general be more complex than the former, and may involve additional kinds of circumstantial dependence, compositional structure, or relational complexities such as polarity. But they surely will not be totally distinct. ⟦strange mark⟧

In section 7 I introduced the phrase "correspondence continuum" to connote the interacting complex of difference correspondence relations we often find connecting representation and represented. However, I equally intended the words to suggest the different kind of continuity arising here: of a full range of variation of type of representational structure.

A simple example will illustrate how continuous these types can be. Modern architectural blueprints used in building construction contain what, to the uninitiated, can be a bewildering range of symbols, ranging from obviously analogue outlines of room shapes, through suggestive icons indicating plumbing and kitchen fixtures, heaters, etc., through slightly stylised icons for electrical outlets, light switches, etc. (with a number of slashes to indicate number of individual outlets, an 'S' to mark whether they are switched, etc.), through general purpose furniture icons with simple inscribed names (desk, bed, etc.), through icons with manufacturer's annotations ("Vermont Castings," "Wolf," etc.), through intermixed sketches, diagrams, and annotations on construction technique, all permeated with arrows, English comments, stamps of approval, scribblings to cancel out parts of the specification, and so on and so forth. That there is a rich variety of representation seems without doubt; that a theoretical scalpel could carve the assemblage into a few neat categories, extraordinarily unlikely.

The moral is unchanged: in variety, detail, and forms of correspondence, current representational practice vastly outstrips current semantical technique. Recognising that most extant theoretical apparatus was developed primarily in service of very particular representational systems employed for logic and metamathematics, we should instead embrace what Ken Olson has suggested:[50] a return to as various and thick a structure of correspondence relations as Peirce ever imagined. Unlike Peirce, however, we can avail ourselves of the full battery of rigorous mathematical methods, axiomatic systems, and so forth, that have been developed since his time. Given such a project, we might even be able to rescue some of the richness of the "semiotic" tradition from what has been perceived to be its vagueness and descriptive complexity.

The fourth and final consequence listed at the beginning of this section has to do with lurking problems in the traditional approach. Those problems, however, arise from fundamental metaphysical questions, and will as such be addressed in the next section.

10 Theories, Models, and Metaphysics

Figure 13 painted a continuum of relations, starting on the left with the linguistic or representational structure under analysis, and progressing in some fashion towards the "real world" on the right. I have suggested that a correspondence theory would provide us with an ability to characterise the relations among the structures comprising this whole, but I have not addressed the question of how one would locate oneself in the resulting continuum. If, as I have suggested, the practice of calling certain relations "syntactic" and others "semantic" is not helpful, is there any other way to distinguish one analysis from another? Or, to put the same question the other way around, can we say anything about traditional approaches? How are they located on this as-yet rather unstructured map?

Four things can be said.

First, if the picture we have been developing is even roughly correct, it predicts that we will encounter structures at various stages across the continuum—relatively more "linguistic" or "syntactic" ones, closer to the primary representational source on the left, others midway across, perhaps having to do with meaning or other semantic (or efficient) uniformities, and others relatively more directly metaphysical or ontological, closer to the full buzzing confusion on the right. That the distinction becomes a matter

[50] «Ref Olson—PhD at Stanford?»

of degree, rather than a binary decision, makes sense of various traditional debates and disagreements. In particular, it is somewhat of a theoretical relief.

To be specific, many people (I am one) have worried about the metaphysical foundations of particular model-theoretic analyses of language,[51] feeling that the proposed model structures reflect, at least in part, the structure of language, not the structure of the world the language is about. For example, consider an analysis (such as a term model) that posits distinct one, two, and three-place relations for various different uses of the verb 'break' (as in "The window broke," "The hockey puck broke the window," and "I broke the window with a hockey puck"). Or imagine an analysis that distinguishes the Pope's saying Mass from the *fact* of the Pope's saying Mass. Or imagine (not hard!) debates about the metaphysical reality of possible worlds, with some people saying that they are real, others saying that they are merely theoretical devices with which to classify language, others claiming that arguments about the reality of semantical constructs miss the point, which is after all to prove various mathematical facts about the linguistic structures themselves. Or suppose someone were to doubt, on metaphysical grounds, the received wisdom that positive and negative facts are on a par, believing instead that this symmetry is a device of language, not a fixture in the world.

If one were to adopt the traditional binary view, then all such questions must be settled one way or the other. I.e., you would have to reject an otherwise appealing semantical analysis if the semantical structures it proposed were metaphysically unconvincing. On the kind of view I am suggesting, however, the whole continuum of possibilities is *exactly what one would expect*. You could accept a term model semantics, for example, but understand it as living rather close to the left hand side, and then ask for further relations to anchor it in, or relate it to states of affairs, further to the right. The structure of the continuum, that is, gives you a way of accepting your fellow theorists' intellectual contributions, even while disagreeing with their metaphysical predilections.

[51] The difficulties are blatant in term models, evident in Kripke style possible world structures, but still apparent, at least to my mind, in the structure of the situation-theoretic universe (Barwise (1986a)).

Second, there are several ways one might locate a particular correspondent structure in a given semantical analysis. For example, it was pointed out early on that much of the semantical contribution of linguistic use arises from circumstances of utterance, not directly from the structure of the sentence used (as in the "I'm right; you're wrong!" example). One of Barwise and Perry's chief points about language[52] is that this property, which they call *efficiency*,[53] is necessary to the proper functioning of communication. It is natural, then, to imagine an analysis of language use that spelled out this circumstantial dependence. It is also easy to imagine, as a semanticist, wanting to avoid the recalcitrant metaphysical problems that arise when you try to map specific vocabulary items onto the world itself (see below). So the following approach might suggest itself: develop a correspondent structure midway between utterances and the world, in such a way that the entire circumstantial dependence of language, up to questions about the metaphysical foundations of vocabulary, has been discharged. The resulting structure is liable to be infinite, but of course that is not a theoretical problem.[54]

This seems a productive way to understand the semantical structures posited both by possible world semantics and situation theory. Needless to say, there are important differences between the two proposals, some of which we can describe: possible world semantics *models* what it calls the interpretation of sentences, whereas situation theory (at least in recent variants) tries to deal with interpretation directly.[x] But the point is to reject as too simplistic the question of whether the structures they each propose are to be viewed as: (i) *the structure of the world*, albeit highly idealised; or (ii) *the structure of language*, albeit decontextualised. Instead, they can both be understood as intermediate analyses.

Third, it is important to dispel a false assumption about how

[52] «Ref»

[53] «Has this been introduced before? If so, refer back; if not, explain?»

[54] John Etchemendy once suggested that the situation-theoretic universe could be viewed in this way (the world of situations, types, states of affairs, etc.—not the language or notation used to describe it): as the world's only non-situated language.

[x] «Or so at least they claimed. Note that I part(ed?) company with this claim of theirs.»

correspondence relations will go, as we move from left to right. As many writers have noted, far more distinctions are made in the syntax of most formal languages than in the model-theoretic structures posited as their interpretations. The most extreme example is the traditional (Fregean) interpretation of all sentences as denoting one of two values: Truth or Falsity. But the general situation is much more common: different spellings with the same content; different procedures designating the same function; etc. Similarly, logical proof theory, defined in terms of syntax (towards the left) pays attention to far more details than does traditional model theory (though of course proof theory does not pay attention to all details, such as to when a formula was written, or to whether parentheses or brackets were used). All of these examples suggest, in general, that correspondence relations will gradually lose information, as they move towards the right, as suggested in figure 17. This assumption is for example embedded in approaches that use initial and final algebras as interpretations for programming constructs.

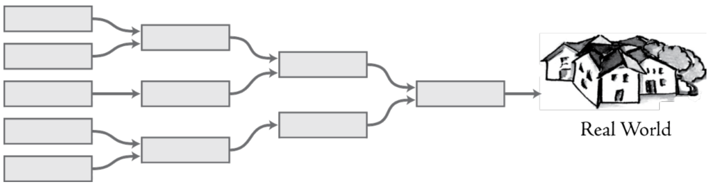


Figure 17: The "losing information" view of semantics

Considerations of circumstantial dependence, however, and some metaphysical arguments, suggest that this neat structure may be an artifact of formal languages, not a general truth of semantics.[55] In the general case, in other words, semantics should not be viewed as a way of moving from fine- to coarse-grained linguistic distinctions. This stance is clearly false if circumstance is ignored: different uses of the word 'I', as we have pointed out so often, can refer to indefinitely many different people, as can 'now' refer to arbitrarily many different times. But more complex phenomena suggest other structures, too. For example, imagine an analysis of natural language, along the lines suggested above, that

[55] Barwise (1986b, p. 331), in fact, defines "formal" languages to be exactly those that are not circumstantially dependent in this way.

ignores different people's sense of the reference of some term 'guilt', say, or 'like'—about which interpersonal agreement is rare. If there is a fact of the matter, when a given person says "She likes feeling guilty," as to what aspect or property of the world is thereby named, then it follows that the *real* connection from utterance to world will discriminate more finely than our chosen semantical analysis.

I choose this example because I can imagine that it would be a serious mistake to try, in the analysis of language, to compensate for such differences by writing them in terms of an explicit parameter for something like "speaker's conceptual scheme"—what I will call **registration scheme**—and then to try to connect such a thing to our previous conception of a "pre-registered" correspondent domain. For some purposes, that is, we may not *want* to capture all the richness of the representation, nor all the richness of the world, nor all the richness of the connection between the two. But this fact still does not allow the conclusion that richness recedes as one moves to the right.

Fourth and finally,[x] there remains the very serious metaphysical question of how any analysis at all is going to deal with the right hand end: the world itself. In fact our continuum seems to suggest that one of the great appeals of the model-theoretic semantical approach—for natural language, AI, and other systems—is that it stops the analysis half-way across the continuum. As suggested above, there are those who worry that the resulting models are still infected with the structure of the languages they purport to analyse, but this has its advantages. Theorists who disagree wildly on the actual structure of the world itself (if that even means anything coherent) can nonetheless agree on a model-theoretic structure. More specifically, one would expect proportionally more agreement—among realists, skeptics, idealists, and theorists of every conceivable metaphysical stripe—to the extent that one's semantic analysis establishes a correspondence to a structure further towards the left. In fact any two people who agreed on an analysis *all the way towards the right* would by definition be of exactly the same metaphysical persuasion; that is what such agreement would *mean*.

[x] «OK, here is the real introduction to O3, metaphysics, etc. … »

The strongest claim I will make about metaphysical grounding will arise in the next and final section, when I return to the semantics of knowledge representation, but a preliminary point can be made here. It has to do with semantics as an instance of theoretical inquiry. To start with, make the following two relatively non-controversial assumptions.

1. Assume that we human theorists, when we use language, are somehow able to refer to the world itself, even if we do not yet know how. I.e., assume something like the most modest form of realism possible: just that there is a world, that we are in it, and that our words somehow enable us to get at it. This is all perfectly compatible with everyone's carving it up in radically different ways, as dictated by nature, nurture, or just plain whim.

2. Assume as well that theories are linguistic vehicles with which we communicate our understanding to our fellow person. Or assume that theories are linguistic entities claimed to be true; for these purposes the difference does not matter.

Once these two assumptions are granted, the following is an immediate conclusion: To the extent that our theories are legitimate instances of language, and thus that we who use or understand them are able to refer to the world, it follows that, as theorists, we do not lack ways of getting to the right hand end of the diagram. I, for example, can get there right this minute with the phrase "this lukewarm cup of coffee to my right." The problem, of course, is that I do not necessarily know various things: not only how it is that I manage to refer to the cup, but also the way in which I have thereby referred to it. So the metaphysical problem for semantical theorists is not one of *referring* to the world by using theoretical language, but rather something closer to the opposite: there is no way of referring to the world *except* by using language. Neurath's boat once again.

This much is obvious. What is important about it is that it is true *all the way across the continuum:* we have no way to refer to the representational structure on the left, or to any intermediating correspondent structure, outside of language either. It only feels more problematic towards the right because it is there that we encounter a natural tendency to want to escape our own particular conceptual schemes, especially if we and the representational structure in question part company. (What he calls "duty" she calls 'guilt.")

This may indeed may be a real limitation: the chances of *completely explaining,* all the way to the right, the semantical interpretation of a system whose conceptual scheme differs radically from one's own, is probably nil. Radical indeterminacy of translation, if there is such a thing, surely has what we might call radical indeterminacy of semantics as a sub-species. But there are more interesting conclusions, as suggested in figure 18.

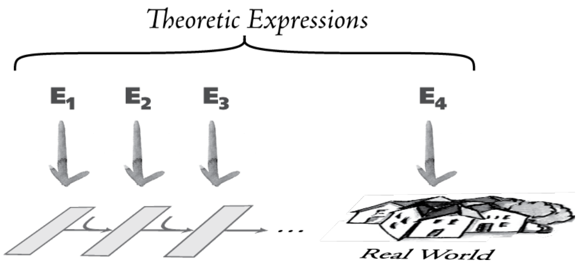


Figure 18: Semantics of theories of correspon-

To the extent that theorist's language and representation overlap on registration scheme, the problems are clearly that much less. This is the happier case, of course, but it has this curious consequence: as analysis moves towards the right, it will *look,* to an outside observer, as if the representation in question is gradually being translated into the theorist's own language—rather on the model of deflationary accounts of truth and reference. I.e., we might say that the noun 'chat' (towards the left) is modelled by the objectified CAT relation (middle), which in turn characterises the set of *real cats* (right). I.e., quotation on the left, reification or nominalization in the middle, and ordinary use on the right. But this is just as it should be; it is predicted by the diagram. There is absolutely no reason to conclude, from this observation, that semantics inherently involves translation.

On the other hand, to the extent that the theorist's registration scheme is his own, it will be so all the way across the diagram. Just because the theorist registers the representational structure itself in terms of a given set of properties and relations (say, as having a particular syntactic form), there is no reason to believe that the representational system registers itself in this way if indeed there is any reason to suppose that it registers itself at all. I.e., if, as I am inclined to suppose, registration involves representation (as well as *vice versa*), then the subject system will register *only* what is to the right; the rest is registered *only for theoretical purposes.*[56] As before, conflict can occur only at the right hand end, but only because that is the only thing that *both* system and theorist register.

In sum, the idea that semantics involves translation is a superficial rendering of the much deeper though perfectly straightforward fact that semantical analysis, like all theoretical investigation, is carried on in language, left through middle through right.[xx]

11 Knowledge Representation Revisited

Although we may seem to have strayed a fair distance from knowledge representation, its demands have been our constant motivation. First, we have seen that the semantical competition between 'representation' and 'knowledge'[x] was merely the tip of a rather large iceberg: without even trying to enumerate an exhaustive list, half a dozen other intentional notions were added to the semantical roster. Second, with respect to appropriate semantical technique, I argued for the prior development of a comprehensive theory of correspondence, and sketched some preparatory philosophical foundations. One way to view this proposed theory is as a branch of semi-mathematics that would immeasurably aid semantics in two ways: by clarifying the semantical project itself, and by providing conceptual vocabulary in terms of which to classify genuinely semantic relations.

On the other hand, I have tried to say plainly that a theory of correspondence would not itself be a theory of semantics, or representation, or knowledge; in fact, in spite of all the ground we have covered, I have said virtually nothing here about the essence

[56] The theorist, of course, can either be us, or else the system introspecting on itself; see Smith (1986).

[x] «Think through the foregoing few paragraphs; do they make any sense?»

[x] «Raises the question: should I here, or in the overall introduction, tie this back to the "From Symbols to Knowledge" Response to Newell & Simon? Probably … »

of any such notions. Even section 9, which tries to sketch some of the structure in which semantics would proceed, still does nothing to resolve this piece of homework. Nor can I do more here. My only intent, by way of a last conclusion, is to make one brief foray in this direction, which will tie the whole analysis back to the primary distinction made at the outset, between representational import and functional role.

The point is simple. I said that functional role and representational import must be coordinated: the agent must be able to act sensibly in terms of what it represents, and (perhaps) represent what it can act sensibly towards. This coordination can be viewed as a kind of "coming together" of knowledge (second factor) and action (first factor). Thus, suppose, knowing the paper is almost over, I reject the lukewarm coffee on my right in favour of a plan that, which it finally is done, I will try some of the Lagavulin in the cupboard. When the time comes, I would like my internal impression that represents the Lagavulin to engender the action of my crossing the room, pouring out a glass, and raising it to my mouth. What is of paramount importance, for our purposes, is the following fact: in the terms of the continuum diagram, this coming together of representation import and action (which is one kind of functional role) must be *all the way to the right.* I want to drink *what is in the world,* not a model or indirect classification of a particularly smoky whiskey, nor a term model of 'Lagavulin' expressions, nor a set-theoretic assemblage of sentences or impressions containing representations of the property of being whiskey. Whatever "stuff itself" is, this much is certain: it is stuff itself towards which my actions must be directed.

This observation, merely a theoretical consequence of the dual facts that action takes place in the world, and that functional role is a kind of action, is the grounds for our sixth and final challenge to the model-theoretic tradition, promised earlier. Because computer systems participate with us in the world—stop our cars, launch our weapons, deliver our mail—it is imperative that our analyses of the representational import of impressions take us all the way to the real world situations towards which the engendered action will be directed. Tooth decay among children will not be reduced by a computer's injecting a mathematical model of fluorine into a set of possible worlds. In order to see the coordination between functional role and representational import, that is, both parts of our two-factor analysis of significance *must reach all the way to the right*. Let's call an analysis that reaches out that far a **grounded** account.

So far, then, the only coordination requirement I will put on theories of full significance is that they be grounded. At least for the moment, that will have to be requirement enough.

References

Barwise, Jon (1986a), "Situations, Sets, and the Axiom of Foundation," Alex Wilkie ed., *Logic Colloquium* 84 Amsterdam: North Holland.

———— (1986b), "Information and Circumstance," *Notre Dame Journal of Formal Logic*, Volume 27 Number 3, July 1986.

Barwise, Jon, and Perry, John (1983), *Situations and Attitudes*, Bradford Books, Cambridge, MA.

Block, Ned, ed., (1981), *Readings in Philosophy of Psychology, Vol 2*, Harvard University Press, Cambridge, MA.

Block, Ned (1985), "Advertisement for a Semantics for Psychology," *Midwest Studies in Philosophy X*, P. A. French, T. E. Uehling and H. K. Wettstein, eds.

Boyd, Richard (1979), "Metaphor and Theory Change: What is 'Metaphor' a Metaphor For?", in A. Ortony, ed., *Metaphor and Thought*, Cambridge University Press, Cambridge, MA, pp. 356–419.

Brachman, Ronald J., and Levesque, Hector J., eds., (1985), *Readings in Knowledge Representation*, Morgan Kaufmann, Los Altos, CA, 571 pp.

Field, Hartry (1977), "Logic, Meaning, and Conceptual Role," *Journal of Philosophy* Vo!. 74.

———— (1978) , "Mental Representation," *Erkenntnis*, Vol. 13.

Fodor, Jerry (1975), *The Language of Thought*, Thomas Y. Crowell Co.: New York. Paperback version, Harvard University Press (1979), Cambridge, MA.

Goguen, Joseph A. and Meseguer, Jose (1984), "Equality, Types, Modules and Generics for Logic Programming," CSLI Technical Report CSLI-84-5, Stanford University, Stanford, CA.

Goodman, Nelson (1983), *Fact, Fiction and Forecast*, Harvard University Press: Cambridge, MA.

Gordon, Michael (1979), *The Denotational Description of Programming Languages: An Introduction*, Springer-Verlag: New York.

Hayes, Patrick J. (1974), "Some Problems and Non-Problems in Representation Theory," *Proc. AISB Summer Conference*, University of Sussex, pp. 63–79. Reprinted in Brachman and Levesque (1985), pp. 3-22.

———— (1977), "In Defence of Logic," *Proc. IJCAI-77*, Cambridge, MA pp. 559–65.

Levesque, Hector (1984), "Foundations of a Functional Approach to

Knowledge Representation," *Artificial Intelligence,* Vol. 23, pp. 155–212.

Lewis, David (1972), "General Semantics," in D. Davison and G. Harman, eds., *Semantics of Natural Language,* D. Reidel, Dordrecht, Holland, pp. 169–218.

Loar, Brian (1982), "Conceptual Role and Truth Conditions," *Notre Dame Journal of Formal Logic,* Vo!. 23(3).

Moore, Robert C., and Hendrix, Gary G. (1979), "Computational Models of Belief and the Semantics of Belief Sentences," SRI International Technical Note 187, Menlo Park, CA.

Moschovakis, Yannis (1984), in *Lecture Notes in Mathematics: Vol. 1103 on Model Theory,* Heidelberg: Springer-Verlag.

Newell, AlIen (1982), "The Knowledge Level," *Artificial Intelligence* Vol. 18 (1), pp. 87-127.

Olson, Kenneth (1985), personal communication at CSLI.

Pylyshyn, Zenon (1978), "Imagery and Artificial Intelligence," in C. W. Savage, ed., *Perception and Cognition: Issues in the Foundations of Psychology, Minnesota Studies in the Philosophy of Science,* Vol. 9. University of Minnesota Press, Minneapolis, pp. 19–55. Reprinted in Block (1981), pp. 170–94.

———— (1984), *Computation and Cognition: Toward a Foundation for Cognitive Science,* The MIT Press/a Bradford Book, Cambridge, MA.

Rosenschein, Stanley J. (1985), "Formal Theories of Knowledge in AI and Robotics," SRI International Technical Note 362 Menlo Park, CA.

Sloman, Aaron (1971), "Interactions Between Philosophy and Artificial Intelligence: The Role of Intuition and Non-Logical Reasoning in Intelligence," *Artificial Intelligence* Vol. 2 pp. 209–25.

———— (1975), "Afterthoughts on Analogical Representation," *Proc. Theoretical Issues in Natural Language Processing,* Cambridge, MA pp. 164–68. Reprinted in Brachman and Levesque (1985) pp. 431–39.

Smith, Brian C. (1982a), *Reflection and Semantics in a Procedural Language,* Technical Report MIT/LCS TR-272, Massachusetts Institute of Technology, Cambridge, MA, 672 pp. See also Smith (1985).

———— (1982b) "Linguistic and Computational Semantics," *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics,* Toronto, Ontario, June 1982.

———— (1984), "Reflection and Semantics in Lisp", *Conference Record of 11th POPL* pp. 23–35, Salt Lake City, Utah. Also available as Xerox PARC Intelligent Systems Laboratory Technical Report ISL-5, Palo Alto, California, 1984.

———— (1985), "Prologue to *Reflection and Semantics in a Procedural Language,"* in Brachman and Levesque (1985), pp. 31–39.

———— (1986), "Varieties of Self-Reference", in *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference,* Morgan Kaufmann, Los Altos, CA. Also available as CSLI-87-7?, Stanford University, Stanford, CA.

———— (1987), "The Semantics of Clocks", *Synthese,* «forthcoming».

———— (forthcoming (a)), *Is Computation Formal?*, MIT Press/A Bradford Book, Cambridge, MA (1987). «update!»

———— (forthcoming (b)), "Representation and Registration". «update!»

———— (forthcoming (c)), "Categories of Correspondence". «update!»

Stich, Steven (1985), *From Folk Psychology to Cognitive Science: The Case Against Belief*, MIT Press/A Bradford Book, Cambridge, MA.

Woods, William A. (1981), "Procedural Semantics as a Theory of Meaning," in A. Joshi, B. Webber, and I. Sag (eds.), *Elements of Discourse Understanding,* Cambridge University Press, Cambridge.

Epilogue

— Were this page been blank, that would have been unintentional —