The Implementation of Procedurally Reflective Languages

Jim des Rivières and Brian Cantwell Smith

July 1984

Intelligent Systems Laboratory
ISL-4

Corporate Accession P84-00070

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

The Implementation of Procedurally Reflective Languages

Jim des Rivières and Brian Cantwell Smith

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.

This is a slightly revised version of a paper that appeared in the Proceedings of the 1984 ACM Symposium on LISP and Functional Programming. This report was also published as Report No. *CSLI-84-9*, Stanford University Center for the Study of Language and Information, July 1984.

CR Categories and Subject Descriptors:

D.3.2 **[Programming Languages]**: Language Classifications - *Extensible Languages, Applicative Languages*, LISP;

D.3.3 **[Programming Languages]**: Language Constructs - *Control structures*;

D.3.4 **[Programming Languages]**: Processors - *Interpreters*;

D.2.6 **[Software Engineering]**: Programming Environments

General Terms: Languages

Additional Key Words and Phrases: Procedural Reflection, Meta-circular Interpreter, Meta-level Control, Level-shifting Implementation, Self-reference.

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 *levels* at which programs are processed, all simultaneously active (in exactly the same way that a traditional program 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 meta-circular interpreters [McCarthy 65, Steele & Sussman 78b], 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 contructs, 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.

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.[2] In general, reflection is a technique whereby a theory *of* a language embedded *within* a language can convey otherwise unavailable power.

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 [Smith 84, Smith & des Rivières 84].

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**.[3] 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 microcode program running at level 2.[4] 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.[5]

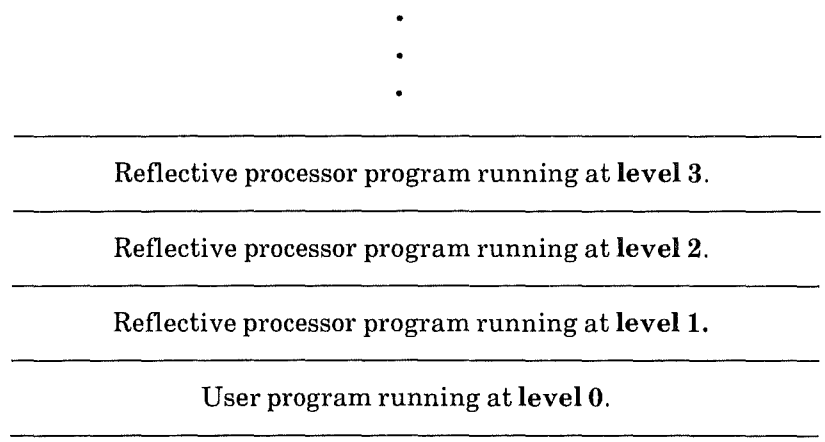

FIGURE 1. The numbering of processing levels in a reflective tower.

The claim that a user's program runs at level 0 is in fact a lie: the whole point of procedurally reflective languages is also to allow user code 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.* 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.[6] Call a processing level *boring* if the only expressions that are processed 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 $\Delta=0$. Normal programs (i.e., standard user programs that don't use any reflective capabilities) will have $\Delta=1$, 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. $\Delta=2$ 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.

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* $\Delta=1$. The existence of such a **G** is a reasonable presumption, since **G** is essentially just a processor for the language 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 [Allen 78, Steele 77a, Henderson 80].

Given **G**, we can show why any reflective program is tractable by induction. The crucial observation is that the overall degree of introspection of an RPP that is running some $\Delta=\mathbf{n}$ program is itself $\Delta=\mathbf{n}$-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 $\Delta\leq 2$ user program will be processed correctly. In general, any $\Delta\leq\mathbf{n}$ program can be run correctly

by **G** provided that **n**-1 levels of genuine RPP are placed in between. This result is depicted in FIGURE 2. (Note that we have talked previously only about a *program's* running at a given level; when we introduced **G** we have described it — an active process, not a program — as running at some level as well. The relationship is this: if **G** is running at level **k**, we mean that the program at level **k** is run by **G** directly, without any higher levels of RPP.)

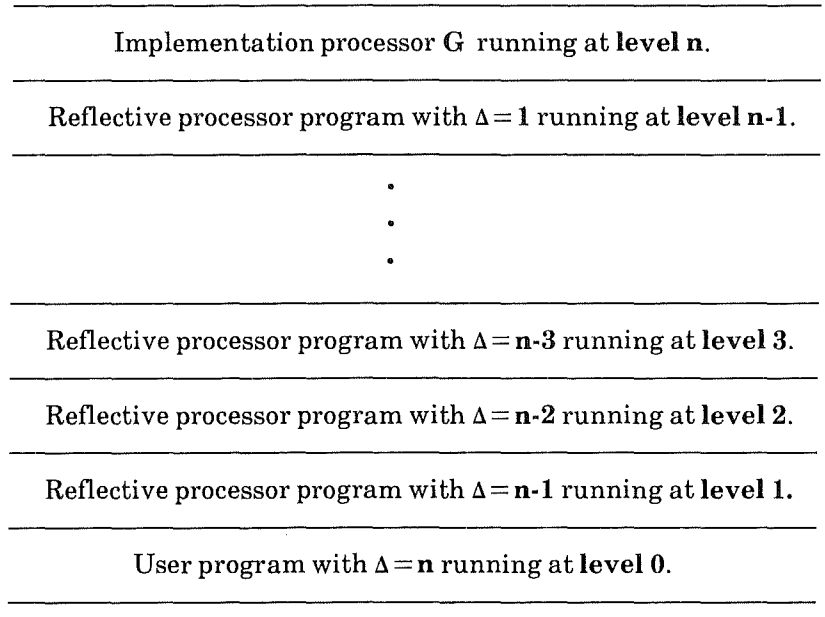

FIGURE 2. How to run a $\Delta = n$ program with a black-box processor that can only handle $\Delta = 1$ programs.

Since it is unlikely that a program's Δ can be determined without processing it, the tractability argument just given doesn't 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. 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 didn't work. More formally, assume initially that $\Delta = 1$, and give the program to **G** to run directly. If **G** detects that the program that it is running has $\Delta > 1$, start the whole computation over again, but this time run it 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 $\Delta > 1$, and that the computation be re-startable.[7] Both of these assumptions are theoretically

reasonable, even though the second isn't practically recommended.

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 had "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 there right from the beginning. In other words, the actual implementation processor would be able to make an *instantaneous shift up* 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 $\Delta=\mathbf{n}$ 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 it will start being run double-indirectly. And so on.[8] Over the course of the computation, in other words, **G'** will gradually climb to higher and higher reflective levels. Although its strategy for shifting levels isn't very sophisticated, **G'** exemplifies the 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 to 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 $\Delta=0$, that 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 is not necessarily 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 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; just what it is to absorb this state in a way so that it can later be rendered explicit, should the need arise, requires some care, 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 $n \to \infty$, 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 only of running $\Delta = 1$ 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 is 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 [Smith 82a] is a reduction-based, higher-order, lexically scoped dialect of LISP whose closest ancestor is SCHEME. 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 *reference* (what an expression *designates, stands for, refers to, names*) and *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 ... x_n]) is abbreviated (f x_1 x_2 ... 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 designates in turn the number one hundred); '$T designates the *boolean* $T; '[1 2] designates the *rail* [1 2]; 'FOO designates the *atom* FOO; '(X . Y) designates the *pair* (X . Y). There are also normal form function designators called *closures*, which have no adequate printed representation. The expressions ''FOO, ''[1], and ''''$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 well 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 one of the two terms 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 meta-circular interpreters discussed in the "Lambda papers" [Sussman & Steele 75; Steele & Sussman 76, 78a, 78b, 80; Steele 76, 77a, 77b], we present in FIGURE 3 the continuation-passing 3-LISP RPP (note: variable names ending in '!' are used, by convention, to indicate that they will always designate normal-form structures).

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 $X \Rightarrow Y$ to mean that X normalises to Y. For example, (+ 1 2) $\Rightarrow$ 3; (PCONS 'A 'B) $\Rightarrow$ '(A . B); ((LAMBDA SIMPLE [X] (* X X)) 4) $\Rightarrow$ 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, which have been

```
1  ..... (define READ-NORMALISE-PRINT
2  ..... (lambda simple [level env]
3  ..... (normalise (prompt&read level) env
4  ..... (lambda simple [result]                                  ; REPLY continuation
5  ..... (block (prompt&reply result level)
6  ..... (read-normalise-print level env))))))

7  ..... (define NORMALISE
8  ..... (lambda simple [exp env cont]
9  ..... (cond [(normal exp) (cont exp)]
10 ..... [(atom exp) (cont (binding exp env))]
11 ..... [(rail exp) (normalise-rail exp env cont)]
12 ..... [(pair exp) (reduce (car exp) (cdr exp) env cont)])))

13 ..... (define REDUCE
14 ..... (lambda simple [proc args env cont]
15 ..... (normalise proc env
16 ..... (lambda simple [proc!]                                  ; PROC continuation
17 ..... (if (reflective proc!)
18 ..... (↓(de-reflect proc!) args env cont)
19 ..... (normalise args env
20 ..... (lambda simple [args!]                                  ; ARGS continuation
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!]                                 ; FIRST continuation
32 ..... (normalise-rail (rest rail) env
33 ..... (lambda simple [rest!]                                  ; REST continuation
34 ..... (cont (prep first! rest!)))))))))

35 ..... (define LAMBDA
36 ..... (lambda reflect [[kind pattern body] env cont]
37 ..... (cont (ccons kind ↑env pattern body))))

38 ..... (define IF
39 ..... (lambda reflect [[premise c1 c2] env cont]
40 ..... (normalise premise env
41 ..... (lambda simple [premise!]                                ; IF continuation
42 ..... (normalise (ef ↓premise! c1 c2) env cont)))))
```

FIGURE 3: The 3-LISP Reflective Processor Program (RPP)

omitted only to shorten the presentation). PROMPT&READ and PROMPT&REPLY issue the system's '**level**>' and '**level**=' prompts, and perform input and output, respectively, but are otherwise innocuous; ↑ and ↓ mediate between a structure and what it designates: (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: (↓(de-reflect proc!) args env cont). 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 unnormalised argument structure (from the original level 0 pair) '[e_1 ... e_n], 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 his program. 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>
```

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 env 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 arbirtary.
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-env catch-cont]
    (normalise
        body
        (bind tag
              ↑(lambda reflect [[answer] throw-env 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 meta-circular processor or RPP can be viewed as an account of a language (or at least of how it is processed) that is 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. This reflective processor was designed to be similar to standard Scott-Strachey continuation-based semantic accounts of λ-calculus based languages (e.g., [Stoy 77, Muchnick 80]). 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 I/O 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 idiosyncracies will not be crucial; the actual code for a 3-LISP implementation is deferred until section 5.

4.1 Level Shifting in Conventional Implementations

Although procedurally reflective architectures are new, the idea of *level shifting* processors isn't. Consider for example an implementation of LISP that supports both interpreted and compiled procedures definitions. In such a system, the non-compiled procedures 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), is 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.

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.

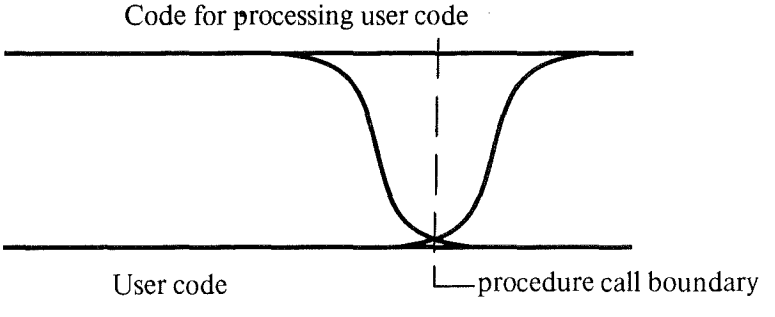

Figure 4: Simple level shifting caused by calls between compiled and non-compiled procedures.

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 ML. 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 (LM); a machine language program to process machine language (MM), and a LISP program to process LISP (LL) (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 a) the one processor program ML, and b) a simple underlying processor **G** that only knows 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 MM — 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 LL.

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 ML) 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 LL, the meta-circular processor program for LISP). The net effect will be that ML will process the code for FOO indirectly — specifically, ML will process LL (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 ML 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 LL 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 two different LISP expressions `(EVAL '(FOO 10))` and `(FOO 10)`.

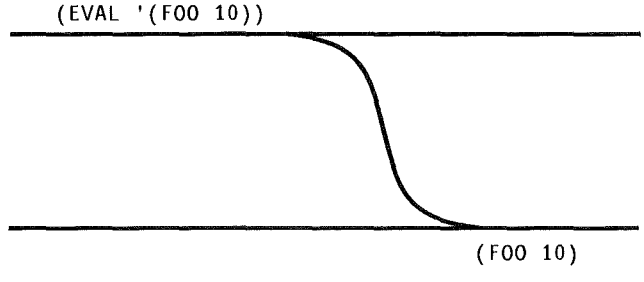

Figure 5: Level shifting caused by calls to EVAL

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 ML — a program, written in machine language, that is a processor for 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.

4.2 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 arbitrary procedures at the level 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 EVAL), making the job of "compiling" NORMALISE much 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 have first 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 meta-circular 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):

```
(COND [(NORMAL '(2ND [10 20])) ... ]
      ... )
(NORMAL '(2ND [10 20]))
    ...                ; various internal steps within NORMAL
(ATOM '(2ND [10 20]))
(RAIL '(2ND [10 20]))
(PAIR '(2ND [10 20]))
(REDUCE (CAR '(2ND [10 20]))
        (CDR '(2ND [10 20]))
        ENV
        CONT)
(CAR '(2ND [10 20]))
(CDR '(2ND [10 20]))
(NORMALISE '2ND ...
(NORMAL '2ND)
    ...                ; various internal steps within NORMAL
(ATOM '2ND)
(BINDING '2ND ...)
...
```

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.

- Primitive procedures, such as 1ST and UP, cannot be decomposed horizontally. Moreover, as the (CONT ↑(↓PROC! . ↓ARGS!)) line of the meta-circular processor shows, 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.
- 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.
- 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.

4.3 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 will typically consist 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: you do not want to end in a dead-end (no further steps possible), and you don't want to encounter a situation where you are climbing a spike without a top.

The board shown in FIGURE 6 was constructed according to the following rules: above every tile numbered x is a sequence of tiles y:

```
x: y    1: 1,2  2: 3,4  3: 1,5  4: 3,5  5: 1,4
```

In constructing a path across the board, only odd-numbered tiles may be stepped on. The best successful path is illustrated by tiles outlined with heavy lines.

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, the game would still be winnable; however, the same can't be said of either the 1-tile or the 5-tile, both of which are unavoidable (notice the insurmountable "spike" of 1-tiles).

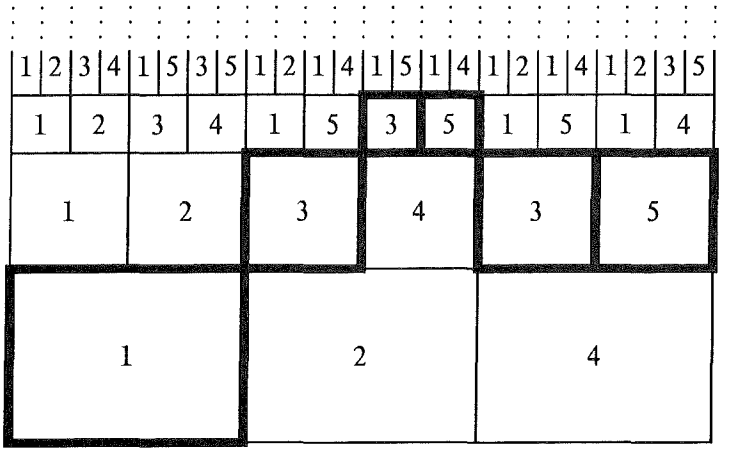

Figure 6: Tiling Game

To implement a reflective language is basically to play a tiling game, where tiles correspond roughly to procedure calls, tiles above another tile are approximately (the horizontal decomposition of) an upwards vertical conversion of the lower tile, and horizontal tiles represent horizontal decompositions. The tiles that can be stepped on are the procedures that have a direct implementation. Like the designer of a tiling game that admits a winning strategy, there is a two-fold challenge: you must carefully select a collection of processing activities that will be implemented directly (corresponding to tiles that can be stepped on), and, for efficiency, you must play the game well, which means coming up with a near-optimal strategy for achieving any $\Delta = n$ (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.

4.4 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 the ppps, the ppcs, the utilities like BINDING, BIND, and NORMAL, and 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 $\Delta = n$ (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.

4.5 When and How to Shift Up

The next important problem is to determine the criteria by which the implementation processor will decide that it is necessary to shift up and 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 ppp's. Not captured at any particular level are the global state of I/O 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).[9] As a result, when a shift up occurs, 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!) ... )`. 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. 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?)
```

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 isn't a definitive answer. Given our particular choice of ppp's, all four of the above 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) **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 doesn't 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; see [Steele & Sussman 76a]); 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 a) the number of bindings in an environment is a (more-or-less) linear function of the static nesting depth of programs, and b) 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. The key implication 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 ppp's always call one another in non-embedded ways. Together with the first observation, it 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 processing the (level 1) RPP processing a (level 0) $\Delta \leq 1$ 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 'REDUCE, 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!) ...) 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.

4.6 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 principle 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 implemention of 3-LISP in 3-LISP (i.e., to write a true meta-circular 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 I/O. 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.

5.1 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!))
        (&&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 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 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 'c1 f) (ex 'c2 f) (ex 'env f)
                   (ex 'cont f))])))
```

5.2 Shifting Up, Shifting Down, and 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 &&CALL, above.

If an expression with $\Delta>1$ 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 didn't write it and therefore don't 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 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; 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. 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:

```
;;; (define &&EXPAND-CLOSURE    ; We don't use this definition!
;;;    (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, we would have (as explained above) to shift back up again immediately, 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. For example, although reflective continuations are well-formed (i.e., `(NORMALISE 'x global (lambda reflect [a e c] (c ↑a)))` normalises to `'[(binding exp env)]`), 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 &&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 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))))
```

5.3 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 "&&" procedure, 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 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 reflection, although tricky, 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 PARC, as part of the Situated Language Program of Stanford's Center for the Study of Language and Information.

Notes

1. We use 'processor' in place of 'interpreter' in order to avoid confusion with the semantic (model-theoretic) notion of interpretation. See [Smith 82a and Smith 84].
2. Exactly the same principle is employed when giving a denotational semantic account of a programming language that has assignment statements: the 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.
3. 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 82a and Smith 84].
4. 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 82b].
5. 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 3).
6. 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 section 4.4, it is the kernel ones that are key to a correct implementation.
7. The re-startability of a computation does not imply that external world side effects (e.g., I/O) are out of the question for a procedurally reflective system. All that would be required is for all interactions with the external world to be remembered by G. Since the restarted computation will merely retrace the steps up to the point that G detected the problem, the computation up to that point could be replayed without having to interact with the external world.

8. We are assuming (not unreasonably) that the point at which it is determined that D>1 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.

9. Although 3-LISP has primitive procedures that "smash" structures, in this paper we will pretend that there aren't any. Without this simplifying assumption, bothersome technicalities would tend to obscure the otherwise straightforward solution. The interested reader is referred to the *Interim 3-LISP Reference Manual* [Smith & des Rivières 84] which contains a correct implementation for the unabridged language.

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.

[Smith, 1984]
Brian C. Smith, "Reflection and Semantics in Lisp", *1984 ACM POPL Conference*, Salt Lake City, Utah, January 1984.

[Smith & des Rivières, 1984]
Brian C. Smith and Jim des Rivières, *Interim 3-LISP Reference Manual*, Xerox Palo Alto Research Center, Intelligent Systems Laboratory Report ISL-1, 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 J. 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 J. 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 J. 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 Rivières 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, 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!))
        (&&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 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)))))
```

```
(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 have to 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 'c1 f) (ex 'c2 f) (ex 'env f)
                    (ex 'cont f))])))
```

;;; The next 6 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)
             (normalise args env
                (lambda [args!]
                   (if (primitive proc!)
                       (cont ↑(↑proc! . ↑args!))
                       (normalise (body proc!)
                                  (bind (pattern proc!)
                                        args!
                                        (environment proc!))
                                  cont ))))))))

(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*
  [↑binding ↑bind ↑rebind ↑de-reflect ↑primitive ↑reflective
   ↑normal ↑length ↑unit ↑member ↑environment ↑double ↑1st
   ↑2nd ↑rest ↑vector-constructor ↑atom ↑pair ↑rail
   ↑handle ↑external ↑normal-rail ↑prompt&read ↑prompt&reply])

(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))] )))

(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))))))
```