SEMANTICALLY BASED AXIOMATICS

Stephen D. Brookes
Carnegie-Mellon University
Department of Computer Science
Pittsburgh
Pennsylvania 15213

ABSTRACT

This paper discusses some fundamental issues related to the construction of semantically based axiomatic proof systems for reasoning about program behavior. We survey foundational work in this area, especially early work of Hoare and Cook on while-programs, and we try to pinpoint the principal ideas contained in this work and to suggest criteria for an appropriate generalization (faithful to these ideas) to a wider variety of programming languages. We argue that the adoption of a mathematically clean semantic model should lead to a natural choice of assertion language(s) for expressing properties of program terms, and to syntax-directed proof systems with clear and simple rules for program constructs. Hoare's ideas suggest that in principle syntax-directed reasoning is possible for all syntactic categories (declarations, commands, even expressions) and all semantic attributes (partial correctness of commands, aliasing properties of declarations, L- or R-values of expressions, proper use of variables, and so on). Semantic insights may also influence assertion language design by suggesting the need for certain logical connectives at the assertion level. This point is obscured by the fact that Hoare's logic for while-programs needed no assertion connectives (although of course the usual logical connectives are permitted inside pre- and post-conditions), but an application of our method to a class of parallel programming languages brings out the idea well: semantic analysis suggests the use of conjunctions at the assertion level. We argue that this method can lead to proof systems which avoid certain inelegant features of some earlier systems: specifically, we avoid the need for "extra-logical" and "non-compositional" notions such as interference checks and auxiliary variables. We also discuss the author's applications of these techniques to other programming languages, and point to some future research directions continuing this work. Although we do not have a completely satisfactory general theory of semantically based axiomatization, and consequently some of our techniques may seem rather *ad hoc* to the reader, we hope that our ideas have some merit.

INTRODUCTION

Our main point in this paper is to develop the argument that axiomatic reasoning should be semantically based, one of the important ideas behind Hoare's early work on proving partial correctness of programs. Although this may sound obvious and in fact most existing proof systems could be claimed to be based on some more or less explicit semantics, we believe that previously proposed proof systems for program properties have sometimes failed to take full advantage of the benefits of well chosen semantic bases. If the underlying semantics is unnecessarily complex, it is more likely that an attempt to design a proof system based upon it will result in errors or undesirable features such as very complicated inference rules with complex side conditions on applicability (or, worse still, unsoundness). We argue that the adoption of a mathematically simple semantic model should lead to a natural choice of an assertion language for expressing program properties, and then to a syntax-directed proof system with clean and simple rules for program constructs. The semantic structure should influence the choice of assertion language, and may even suggest the need for logical connectives such as conjunction at the assertion level, although this is not needed in Hoare's original proof system for while-programs because of their especially simple semantics.

To be more precise, we mean that firstly, when desiring to reason about a certain class of program properties (e.g. partial or total correctness, deadlock-freedom, etc.), one should begin by formulating a semantic model for the programming language which is *adequate* for that program property. By adequacy we mean that the semantics must be able to distinguish between program terms which can be used in some program context to induce different program behaviors. This is one half of the full abstraction definition [17]: full abstraction requires that the semantics should distinguish between terms if and only if they can induce different program behaviors in some program context. Thus, a fully abstract semantics would certainly be adequate; however, full abstraction may be difficult to achieve for certain languages and classes of properties (see [15,23] for instance), while adequacy is rather easier to achieve and suffices for accurate support of syntax-directed reasoning, since it permits the replacement of semantically equal terms inside a program without affecting the overall properties of the program.

Secondly, the semantic model should have a clean mathematical structure. For instance, the standard partial function semantics for while-programs is certainly clean and simple and is at the right level of abstraction to be adequate for partial correctness or, for that matter, for total correctness. Thirdly, the very structure of the semantic model should guide the choice of assertion language(s) for expressing properties of program terms. We illustrate this last idea with examples, although as yet we do not have a completely worked out general theory which would describe for

an arbitrary semantic model how to construct a corresponding assertion language. We have more to say on this in the conclusion, where we also draw attention to other work in this direction.

A particularly important facet of this last idea is that the structure of a semantics may necessitate the use of logical connectives in an assertion language; while it may seem trite to a logician to argue for the inclusion of logical connectives in what is, after all, a logical formalism, Hoare's logic for while-programs contained no explicit connectives. In that case, there was actually no need: a simple semantic justification can be given that shows that one does not need to form (for instance) conjunctions of Hoare logic assertions in order to obtain a complete proof system. However, this property need not hold for more intricate programming languages, and it certainly fails with parallel programs. Again this point is brought out fully by our example. We also point to some directions for further investigation. Firstly, however, we give some historical remarks to set the scene.

HOARE'S LOGIC

The ideas of using axiomatic techniques to reason about program behavior go back to the pioneering work of Floyd [13] and Hoare [14]. In 1966, Hoare proposed an axiomatic basis for computer programming, in a paper of that title. The main idea was very simple but powerful: to use partial correctness assertions (pca's) to specify program properties, and to design a proof system in which pca's about compound programs can be deduced from pca's about their syntactic constituents. In popular terms, Hoare proposed a "syntax-directed" logic for partial correctness. A partial correctness assertion takes the form $\{P\}C\{Q\}$, where P and Q are *conditions* and C is a program. This is intended to be interpreted informally as saying that if C is executed from an initial state satisfying P, then if execution terminates, it does so in a final state satisfying Q. The conditions used by Hoare were drawn from a first order logical language for arithmetic, a natural enough choice since the expressions of his programming language denoted integers and boolean values.

In Hoare's proof system there are axioms for each atomic form of program (e.g., assignment) and inference rules for each program construct. For instance, the following *while*-rule from [14] is well known by now:

$$\frac{\{P \,\&\, B\}C\{P\}}{\{P\}\text{while } B \text{ do } C\{P \,\&\, \neg B\}}$$

This rule allows us to deduce a (special type of) pca about a while-loop from a (special kind of) pca about the loop body. In addition, and what might seem incongruous if one is looking for a purely syntax-directed proof system, Hoare's

proof system included a so-called *rule of consequence*:

$$\frac{P' \Rightarrow P \quad \{P\}C\{Q\} \quad Q \Rightarrow Q'}{\{P'\}C\{Q'\}}$$

This rule, applicable to all while-programs C (not just to loops), allows us to "strengthen" a pre-condition and "weaken" a post-condition. Note also that some such rule is necessary in any case because of the special restrictions on the assertions deducible from the syntax-directed rules; without some general rule like this, one would only be able to prove assertions about loops with the fixed format $\{P \,\&\, B\}$while B do $C\,\{P \,\&\, \neg B\}$, and one would be unable to manipulate conditions inside assertions.

Thus, it is sometimes stated that Hoare's proof system has two parts: a syntax-directed part containing the axioms and rules for the programming constructs, and a "logical part" containing (at least) the rule of consequence. A "logical" rule is applicable to all programs in the programming language, whereas a syntax-directed rule is only applicable to programs built using a specific construct.

SOUNDNESS AND COMPLETENESS

Cook [11] established the soundness of the Hoare system, and also gave what is now a standard definition of completeness for program logics ("relative completeness"). It is useful for our purposes to summarize the essential ideas. Fuller details are in [2] and [11].

Firstly, one needs to make the reasonable assumption that the condition language be chosen to be sufficiently powerful to contain all intermediate conditions needed during a proof. Cook therefore defines an "expressivity" criterion. Bearing in mind that the programming language in question here (while-programs) has a partial function semantics, expressivity boils down to the following property: for every condition P in the condition language and every program C, there must be a condition Q in the condition language characterizing the set of states

$$\{\, s \mid M(C)s \text{ is defined and satisfies } P \,\},$$

where $M(C)$ is the partial function denoted by C. Loosely speaking, expressivity amounts to closure under (semantic) weakest pre-condition (Dijkstra [12]). An equivalent formulation can be given in terms of strongest post-condition (Clarke [9,10]). The first order language for arithmetic used by Hoare is certainly expressive for his application, and we will not focus on this issue further at this point.

Completeness in the standard sense is clearly impossible, given the well known fact that the validity problem for pca's is undecidable ([2]): there is no effective

procedure for testing the validity of partial correctness assertions. This holds even if the condition language is trivially small, as long as it contains **true** and **false**, so that halting problems become expressible as pca's. It is therefore impossible to obtain a complete proof system in the usual sense of completeness, since the set of provable assertions would then be recursive, while the set of valid pca's is recursively enumerable and not recursive. Cook wanted to isolate any incompleteness due to the syntax-directed proof rules and axioms. Correspondingly, he specified that Hoare's logic should be regarded as "relatively" complete if the following condition holds: for every valid pca there is a proof of that pca in which we are allowed to use as axioms or unproved assumptions any valid *condition*. The idea is to allow the proof system to access an "oracle" able to answer validity questions about conditions (and implications between conditions). The primary place where this oracle is useful, of course, is the rule of consequence.

Cook proved the following completeness theorem: for any expressive condition language, Hoare's proof system for while-programs is relatively complete. The proof given by Cook is itself illuminating. He showed that, for every valid pca $\{P\}C\{Q\}$, we can find an assignment of pre-conditions and post-conditions to the subterm occurrences of C which can be used in a natural, syntax-directed manner, to prove $\{P\}C\{Q\}$. Since a particular subterm may occur several times inside a given program, and in general each occurrence serves a different purpose semantically, one needs to be able to say something different about each occurrence; hence the reference to subterm *occurrences* rather than simply to subterms. [This preoccupation with subterm occurrences rather than subterms will recur throughout this paper.] Let $\prec$ denote the relation "is an immediate subterm occurrence of"; the transitive closure of this relation is the subterm occurrence relation, denoted $\prec^*$; at the moment we will only refer to subterms which are commands, although of course subterms may be expressions or declarations. If we let *pre* and *post* be the functions mapping subterm occurrences of C to pre- and post-conditions respectively, the idea is that the set

$$\{\{pre(C')\}C'\{post(C')\} \mid C' \prec^* C\}$$

can be used in a syntax-directed manner to build a proof of $\{P\}C\{Q\}$. The choice of the functions *pre* and *post* obviously depends on P, C, and Q.

To be precise about the requirements on these *pre* and *post* functions, and explain exactly what is intended by "using" this set of pca's in a syntax-directed manner, one would need to provide a collection of constraints. Typical of these is the constraint imposed by a subterm occurrence built by sequential composition: if $C' = C_1; C_2$ is a subterm occurrence of C we require that

$$pre(C') \Rightarrow pre(C_1)$$
$$post(C_1) \Rightarrow pre(C_2)$$
$$post(C_2) \Rightarrow post(C').$$

These constraints allow a proof of

$$\{\, pre(C')\,\}C'\{\, post(C')\,\}$$

from the pair of assertions

$$\{\, pre(C_i)\,\}C_i\{\, post(C_i)\,\}, \quad i = 1, 2$$

by using Hoare's rule for sequential composition and the rule of consequence. There are similar constraints on *pre* and *post* for the other program constructs. In each case, the constraints allow the deduction of the assertion $\{\, pre(C')\,\}C'\{\, post(C')\,\}$ from the corresponding assertions about the (immediate) syntactic subterms of C' and the rule of consequence. This formalizes the notion of a "standard" method of syntax-directed proof.

The assignment of pre- and post-conditions to the subterm occurrences of C in order to prove $\{\, P\,\}C\{\, Q\,\}$ is known as a "proof outline", for obvious reasons. It is important that a single pca suffices for each subterm occurrence; this is the crucial property that enables us to achieve (relative) completeness with a proof system in which each inference rule has a single premise for each immediate syntactic component of the program appearing in the conclusion to the rule: i.e. the rules have fixed finite numbers of premises whose structure directly corresponds to the syntactic structure of the conclusion.

Because of the close connection between the idea of weakest pre-conditions and the construction of the *pre* and *post* functions in Cook's theorem, Cook's proof of relative completeness can be paraphrased as showing that weakest pre-conditions can be used in a straightforward way to prove any valid pca, provided we assume that all necessary reasoning inside the condition language can be carried out. Dijkstra [12] showed that a syntactic definition of weakest pre-conditions can be given for while-programs. Clarke [10] established the connection between Cook's notion of expressibility and the ability of the condition language to contain weakest pre-conditions and/or strongest post-conditions.

LOGICAL RULES AND CONNECTIVES

To a logician there might appear some *ad hoc* features in Hoare's "logic" for while-programs. If pca's are the objects of proof (*cf.* formulas or theorems), where are the logical connectives? One never forms the conjunction of two pca's in Hoare's

logic, nor the disjunction, and certainly not the negation of a pca! Apart from the so-called logical rules, traditional logical connectives appear to play a very minor role in Hoare's proof system. Certainly logical combinations of conditions are manipulated (e.g. in the rules for loops and conditionals, and in the rule of consequence), but the same is emphatically not true for pca's.

The role of the rule of consequence in the completeness property for Hoare's logic is crucial. Other sound logical rules, such as

$$\frac{\{P_1\}C\{Q_1\} \quad \{P_2\}C\{Q_2\}}{\{P_1 \,\&\, P_2\}C\{Q_1 \,\&\, Q_2\}}$$

(or, for that matter, the similar rule involving $\vee$) are not needed to achieve completeness, although it could be argued that they are useful pragmatically. Note also that this rule is not really related to the familiar &-introduction rule of natural deduction, despite its superficial resemblance in that it introduces conjunction in both pre- and post-conditions: the assertion $\{P_1 \& P_2\}C\{Q_1 \& Q_2\}$ does not behave logically as a conjunction of assertions, since the fact that C satisfies it does not logically imply that either of the assertions $\{P_i\}C\{Q_i\}$ is also valid. Of course, in propositional or predicate calculus each of ϕ and ψ is a logical consequence of $\phi \& \psi$. Thus, even rules such as this which superficially seem to involve connectives used with assertions about the same program are not truly treating assertions as objects of a boolean algebra.

Yet the rule of consequence is in reality a disguised form of *modus ponens*. If we define the obvious notion of implication for pca's, i.e. that

$$\{P\}C\{Q\} \Rightarrow \{P'\}C\{Q'\} \quad \Leftrightarrow \quad (P' \Rightarrow P) \,\&\, (Q \Rightarrow Q'),$$

the connection with *modus ponens* becomes clearer. Note, nevertheless, that this "implication" is not a true logical connective on pca's, because it only applies to pca's about the same program. If we reformulate Hoare's ideas without using sugar, writing

$$C \models (P, Q) \quad \text{for} \quad \{P\}C\{Q\}$$

and writing $(P, Q) \Rightarrow (P', Q')$ for $(P' \Rightarrow P) \;\&\; (Q \Rightarrow Q')$, we have for the rule of consequence:

$$\frac{C \models (P, Q) \qquad (P, Q) \Rightarrow (P', Q')}{C \models (P', Q')}$$

This merely syntactic reformulation of the rule involves less syntactic sugar and, I feel, emphasizes better the logical structure of the rule. We will see later that this version generalizes to other settings.

The reason that no logical connectives (other than implication) on assertions were necessary is summarized as follows. In order to prove a (single) pca about a

program C, we can find an assignment of pca's to the syntactic subterm occurrences of C from which a proof can be constructed. This is the content of the "proof outline theorem" as described earlier. If, on the contrary, one might have needed several assertions for each subterm of C, it might have been necessary to allow conjunction of pca's (rather than allowing rules with arbitrarily many premises). Crucially for while-programs, we can get by with a single assertion about each piece of a program.

However, there is no reason why this should hold when we move to more complicated programming languages. Firstly, if a language has a more complicated semantic model (than partial functions from states to states) there is no particular reason to suppose that the pca format should lend itself naturally to syntax-directed reasoning. Essentially, the problem is that pca's may no longer have a structure that "fits" the semantic structure adequate for partial correctness. For example, in [8] we reported on a syntax-directed proof system for a simple block structured language with aliasing introduced by declarations, and we found indeed that the pca format needed to be modified to allow a good fit with semantic structure.

In addition, there may be quite different notions of program correctness which are of interest, and then it is not always most natural to use pca's to express program properties. By way of example, again from [8], when reasoning about the correctness of block structured programs in a setting where aliasing may occur among program variables, it was found necessary to axiomatize the (purely declarative) aliasing properties of declarations in addition to their imperative effects; although the aliasing properties were conveniently expressed in a notation which superficially resembles that of pca's, the "pre- and post-conditions" were drawn from a very simple language solely chosen to allow succinct descriptions of aliasing relationships and did not need the full power of a language for arithmetic. For another example, in which a much more radical departure from the pca format is suggested by the semantic structure, see the application to parallel programs in the next section of this paper.

Finally the need for connectives on assertions should be re-examined for an application to more complicated languages. Again, the parallel programming example brings this out well, and we now examine this in more detail.

AN EXAMPLE

To illustrate our points more concretely, we summarize the application of our ideas to a simple parallel programming language, essentially the language discussed by Owicki in her thesis [19] and in the paper [20]. We provide here a condensed

development of this work; a full presentation appeared in [6]. Many of the details are suppressed to avoid excessive duplication of effort.

The programming language is a standard while-loop imperative language to which we add a parallel composition $C_1 \parallel C_2$ and a conditional critical region **await** B **then** C, in which the body C must be executable as a single atomic unit. We regard assignments and boolean expression evaluations as atomic. [In fact, it is reasonable to constrain the body of a critical region to be a finite sequence of assignments, although this is not crucial to our discussion.] The interpretation of parallel composition will be nondeterministic interleaving of atomic actions of the two parallel processes C_1 and C_2 until both have terminated. A conditional critical region **await** B **then** C can only be executed when B is true, and its effect is to perform all of C without allowing interruption from any other (parallel) process. When its test condition is false, an await command is unable to progress and must, as its name suggests, wait for the state to be changed (by another program executing in parallel) to one satisfying the condition. When a command is unable to progress but has not yet properly terminated (typically because of an await) it is said to be deadlocked. Thus, for instance, the effect of

$$x{:=}0;[(\textbf{await } x = 1 \textbf{ then } y{:=}2) \parallel x{:=}1]$$

is to (finally) set x to 1 and y to 2; on the other hand, the effect of

$$x{:=}1;[(\textbf{await } x = 1 \textbf{ then } y{:=}2) \parallel x{:=}0]$$

is either to deadlock (with x set to zero) or to terminate with $x = 0$ and $y = 2$. Full semantic details appear in [6].

It is well known that for a treatment of partial correctness and deadlock this language requires a more sophisticated semantic model than the obvious modification of state-transformations to relational semantics. Using the operational presentation of Hennessy and Plotkin [15] it is convenient to describe the semantics of this language in terms of the computations of an abstract machine whose configurations have the form $\langle C, s\rangle$ (where C is a command and s is a state), and whose transitions (one-step of a computation) involve a single atomic action and are described by a family of transition relations $\rightarrow^{\alpha}$ (where α ranges over the set of atomic actions). A transition of the form $\langle C, s\rangle \rightarrow^{\alpha} \langle C', s'\rangle$ represents the ability of C, in state s, to execute the atomic action α and that immediately afterwards the state is s' and the remaining command is C'. There are two types of configuration from which no transition is possible: deadlocked and successfully terminated configurations. A full semantic description of the language gives definitions of the transition relations and of two sets *DEAD* and *TERM* of configurations. The partial correctness and deadlock behavior of a program C can then be described as a function

$$M(C) : S \rightarrow (\mathcal{P}(S) \times \mathcal{P}(S))$$

which is itself presentable in the following form, by means of two auxiliary functions

$$\begin{aligned}
&\mathcal{S}(C) : S \to \mathcal{P}(S) \\
&\mathcal{S}(C)s = \{ s' \mid \langle C, s\rangle \to^* \langle C', s'\rangle \in \mathit{TERM}\} \\
&\mathcal{D}(C) : S \to \mathcal{P}(S) \\
&\mathcal{D}(C)s = \{ s' \mid \langle C, s\rangle \to^* \langle C', s'\rangle \in \mathit{DEAD}\} \\
&\mathcal{M}(C)s = \langle \mathcal{S}(C)s, \mathcal{D}(C)s\rangle
\end{aligned}$$

Here $\to$ denotes the one-step transition relation, so that $\langle C, s\rangle \to \langle C', s'\rangle$ represents that for some α we have $\langle C, s\rangle \to^\alpha \langle C', s'\rangle$. We use $\to^*$ for the transitive closure of this relation. Thus, $\mathcal{S}(C)s$ is the set of states in which some finite sequence of transitions may terminate from the initial configuration $\langle C, s\rangle$. However, it is impossible to construct this function in a purely denotational, syntax-directed manner. The reason is that we are ignoring intermediate states and the potential for interference between parallel processes. As Hennessy and Plotkin established, it is necessary instead to use a semantics based on a more intricate structure (resumptions). Ignoring for the moment the fact that the resumptions are recursive objects from a recursively defined domain, we can design a semantics

$$\begin{aligned}
&\mathcal{R} : \mathbf{Com} \to R \\
&R = [S \to \mathcal{P}^*(R \times S)] \\
&\mathcal{R}(C)(s) = \{ \langle \mathcal{R}(C'), s'\rangle \mid \langle C, s\rangle \to \langle C', s'\rangle \}
\end{aligned}$$

We use here $\mathcal{P}^*$ to denote a simple variant of the finite powerset constructor in which there are two versions of the empty set, which we denote $\bullet$ and $\circ$, representing respectively termination and deadlock. Thus, if $\mathcal{R}(C)(s) = \bullet$ we say that C has terminated in state s, and similarly for deadlock. In this structure it is important to note that the sets $\mathcal{R}(C)(s)$ are always finite, and indeed that each member of such a set is the result of a unique atomic action (occurrence) from the text of C which is "enabled" in state s. Again, we refer the reader to [6] for more details. The rigorous mathematical justification for the use of recursively defined domains here is not germane to the paper, although of course justification is necessary for the semantic definitions to make sense.

Now, in the same way that Hoare's syntax for pca's "fits" the semantic structure $[S \to S]$, once we have chosen a syntax for conditions describing S, we may design an assertion language for the structure R. Let ϕ be an assertion describing a resumption r. We need in ϕ to be able to describe, given some information about an initial state s (i.e., given a condition P), whether or not (the command whose resumption is) r deadlocks or terminates, and if not, some information about each of the possible results of the atomic actions enabled in s. Each of these results requires another assertion about a resumption (describing the resulting command) and a "post"-condition describing the resulting state. Thus we are led, with some

syntactic sugaring, to the following "grammar" for ϕ:

$$\phi ::= P\circ \mid P\bullet \mid P\sum_{i=1}^{n}\alpha_i P_i\phi_i$$

We have chosen to include atomic actions in the syntax of assertions purely for ease of relation to other proof systems, and we have chosen a linear notation for the general form of assertion because it resembles very closely Milner's format for synchronization trees [16]; in fact, it is convenient to think of an assertion as a synchronization tree with conditions before and after each of its arcs. Our aim will now be to produce a proof system for establishing properties of the form $C \models \phi$. The interpretation of such a property is, of course, closely based on the semantics.

Now that we have an assertion language whose structure closely models the semantic structure of the objects it describes, it is clearly going to be possible to reason about these objects in a syntax-directed manner, just as the denotational semantics builds meanings of commands in a syntax-directed manner. In fact, just as there is a semantic operation $\|$ on resumptions such that

$$\mathcal{R}(C_1 \| C_2) = \mathcal{R}(C_1) \| \mathcal{R}(C_2)$$

we can introduce a "semantic connective" $\|$ on assertions with the intended property that whenever ϕ_1 describes C_1 and ϕ_2 describes C_2, then also $\phi_1 \| \phi_2$ describes $C_1 \| C_2$. The required definition, taken from [6], is for the "base cases":

$$\begin{aligned}(P\bullet) \| (Q\bullet) &= \{P \,\&\, Q\}\bullet\\ (P\bullet) \| (Q\circ) &= \{P \,\&\, Q\}\circ\\ (P\circ) \| (Q\circ) &= \{P \,\&\, Q\}\circ .\end{aligned}$$

For $\phi = (P\sum_{i=1}^{n}\alpha_i P_i\phi_i)$ and $\psi = (Q\sum_{j=1}^{m}\beta_j Q_j\psi_j)$,

$$\phi \| \psi = \{P \,\&\, Q\}(\sum_{i=1}^{n}\alpha_i P_i[\phi_i \| \psi] + \sum_{j=1}^{m}\beta_j Q_j[\phi \| \psi_j]).$$

Given this definition, which we regard as constituting a logical characterization of $\|$ as a connective on assertions, we can use the following proof rule for reasoning about parallel programs:

$$\frac{C_1 \models \phi_1 \quad C_2 \models \phi_2}{[C_1 \| C_2] \models [\phi_1 \| \phi_2]}$$

Although we have not given the details here, the soundness of this rule is obvious, because the definition of $\phi \| \psi$ is essentially a rephrasing of the semantic clause defining $\mathcal{R}(C_1 \| C_2)$ from $\mathcal{R}(C_1)$ and $\mathcal{R}(C_2)$. Since the assertion language and rules

are semantically based, the soundness proofs are made by appeal to the semantic definitions.

Using syntax-based rules of the above kind for the programming constructs, it is straightforward to design a simple proof system for properties of the form $C \models \phi$, in which the premises in each rule involve assertions about the principle subterms of the conclusion.

At this point, one might ask if the assertion language and proof system are sufficiently powerful. The answer is no, if we want to achieve the desired completeness properties. The first reason is that, as with Hoare's logic for while-programs, we need an analogue of the rule of consequence to allow us to manipulate conditions. There is in fact a very natural generalization of the rule of consequence, which we will embody as a form of *modus ponens*. Firstly, it is possible to define an *implication* on assertions: implication $\Rightarrow$ is characterized by the properties:

$$\begin{aligned} (P\bullet) &\Rightarrow (Q\bullet) \quad \Leftrightarrow \quad (Q \Rightarrow P) \\ (P\circ) &\Rightarrow (Q\circ) \quad \Leftrightarrow \quad (Q \Rightarrow P) \\ (P \sum_{i=1}^{n} \alpha_i P_i \phi_i) &\Rightarrow (Q \sum_{i=1}^{n} \alpha_i Q_i \psi_i) \quad \Leftrightarrow \quad (Q \Rightarrow P)\ \&\ \bigwedge_{i=1}^{n} [(P_i \Rightarrow Q_i)\ \&\ (\phi_i \Rightarrow \psi_i)] \end{aligned}$$

Note that we use $\Rightarrow$ to denote assertion implication and the usual implication on conditions; the context makes it clear which is intended. It should be clear that this definition of implication is the obvious extension to our more highly structured assertion language of the (implicit) notion of implication for ordinary pca's as described earlier. The analogue in this setting to the rule of consequence is:

$$\frac{C \models \phi \qquad (\phi \Rightarrow \psi)}{C \models \psi}$$

Even with the inclusion of this rule, there is not yet any analogue of the Cook proof outline property. There is a second reason for incompleteness. There are simple examples of commands C and assertions ϕ such that $C \models \phi$ is valid, but is not deducible by first proving a *single* assertion for each syntactic subterm occurrence of C. A rather elementary example is:

$$[x{:=}x+1 \parallel x{:=}x+2].$$

No pair of single assertions (one each) about $x{:=}x{+}1$ and $x{:=}x{+}2$ can be combined to prove (the assertion which states) that this program increases x by 3. Instead, in this case, we need to be able to make *two* assertions about each subterm occurrence

(one for use when the term is executed before the other, and one after). The general scheme is that one needs in principal to allow arbitrary finite conjunctions of assertions about each subterm occurrence. Thus, we can recover the "one assertion for each subterm" property by throwing in conjunction at the assertion level! (This also necessitates a careful axiomatization of the interaction of conjunction with parallel composition of assertions, as discussed in [6]. It is also necessary to specify the obvious implicational properties of conjunctions.) If we do this, as shown in [6], we obtain the expected results: a sound and relatively complete proof system. The point to emphasize here is that semantic considerations have led us to include conjunction in the assertion language.

The analogue of Cook's proof outline theorem is then: for every valid assertion $C \models \phi$ there is an assignment *assert* of assertions to subterm occurrences of C such that $C \models \phi$ is deducible in a standard manner from the premises

$$\{ C' \models \mathit{assert}(C') \mid C' \prec^* C \}.$$

Again, the assignment of assertions to subterms must satisfy certain requirements for this standard deduction to be possible. For instance, given a subterm occurrence $C_1 \parallel C_2$ of C, we require that

$$(\mathit{assert}(C_1) \parallel \mathit{assert}(C_2)) \Rightarrow \mathit{assert}(C_1 \parallel C_2).$$

As with the proof outline constraints, each type of syntactic construct imposes a constraint on the assert assignment. Fuller details will be given in an expanded version of [6]. The main point is that assertions about a compound command should be deducible from assertions about its immediate subterms.

It should be noted that the use of conjunction means that we do not need recourse to auxiliary variables; and that the careful definition of parallel composition of assertions was made in order to avoid the need for interference-freedom checks [19,20]. Both of these points are elaborated in more detail in [6] and [7]. It is in avoiding recourse to these rather extra-logical features that we see the principal advantages conferred by our approach.

GENERALIZING HOARE'S LOGIC

We have surveyed early developments in axiomatization and given an example which we believe generalizes the important ideas appropriately to a significantly more complicated programming language. Although we would not claim to have worked out a complete theory of semantically based axiomatization applicable to all possible programming languages and classes of program properties, we feel

that several essential ideas are suggested by our experience. The following points summarize these ideas well.

- Assertion language(s) should be designed to fit semantic model(s).
- Syntax-directed reasoning is (in principle) feasible for all syntactic classes.
- Logical connectives should be included in an assertion language if the semantic properties imply their utility.

The first point was, as we have already remarked, certainly satisfied by Hoare's choice of syntax for pca's, in that a pca contains two conditions, one for the initial and one for the final state. The other two points are perhaps not so clearly visible in Hoare's while-program logic, since the principal topic of his work was the axiomatization of the partial correctness properties of commands alone; there was no need then to axiomatize expressions or declarations, and no need for conjunctions of assertions. Indeed, to some extent, the syntactic sugar used in the pca format obscures the logical structure.

As a point often taken for granted, note that even the choice of *condition* language ought to be influenced by semantic properties. In particular, we know that while-programs suffice to describe all partial recursive functions on the integers (assuming that all expressions are integer-valued). Moreover, if a condition language contains the usual logical connectives (a natural enough property!) and contains conditions of the form $I = E$, where I is a program variable or identifier, and E is an (integer-valued) expression, any finite state (i.e., any finite function from a finite set of identifiers to integers) can be described uniquely by a "characteristic condition", of the form

$$I_1 = v_1 \ \& \ I_2 = v_2 \ \& \ \ldots \ \& \ I_k = v_k.$$

Since programs describe partial recursive functions, the (sets of states described by) conditions must be closed under image and pre-image of partial recursive functions if we are to be able to express all necessary intermediate conditions. This conclusion is obviously related to the results established in [3], where an analysis is given of the suitability and expressive power of recursive and recursively enumerable condition languages.

The second point made above is that syntax-directed reasoning is possible for any syntactic category, not solely for programs. This was already implicit in Hoare's early work [14], but I do not believe that later developments took up the idea to its fullest extent. For one example, some early work on axiomatizing languages with declarations as well as commands utilized the pca format $\{ P \} C \{ Q \}$ again,

but with the pre- and post-conditions also expressing declarative properties such as aliasing relationships. It is conventional to give semantics by means of separate (but related) environments and stores, as for example in Stoy [24] and in Strachey [25]. The meaning of a command is then a function from environments to partial store-transformations, and a declaration denotes an environment-transformation. When this is done, the pca format no longer fits as well: from the format of a pca for commands it appears that we will have to keep *proving* that the command has no effect on the environment, because the post-condition mentions properties of the environment. But only declarations change the environment, so it is actually more convenient to design two proof systems in tandem: one for declarations and one for commands. The assertions for commands should involve two separate pre-conditions and a single post-condition, with a pre-condition for the environment and one also for the store. A detailed exposition of such an approach can be found in [8].

Next we expand on the idea that semantic structure should influence the design of assertion languages and proof systems. We attempt a general definition of what a semantically based, syntax-directed proof system should be.

In a denotational semantic definition for a programming language the meaning of a term is constructed from the meanings of its parts. Given the abstract syntax of a programming language, in particular the set of syntactic *types* (e.g. **Com**, **Exp**, **Ide**, **BExp**) and the syntactic constructors (e.g., ";" of type $\mathbf{Com} \times \mathbf{Com} \to \mathbf{Com}$), a denotational semantics consists of a family of semantic functions, (usually) one for each type, each mapping terms of a particular type into meanings appropriate for that type. These functions are usually presented by a structural induction on the syntax. It is usual to pick out a collection of semantic domains, one for each type, so that if τ ranges over the syntactic types, we can write D_τ for the semantic domain appropriate for type τ and define a semantic function $M_\tau : \tau \to D_\tau$. The denotational condition that a term's meaning is determined from the meanings of its subterms then becomes reflected as follows. For each program construct op of type $\tau_1 \times \cdots \times \tau_n \to \tau$ there is a semantic operation F_{op} which constructs the meaning $M_\tau(op(t_1, \ldots, t_n))$ from the meanings $M_{\tau_i}(t_i)$. With the usual notion of a subterm occurrence, and of immediate subterm occurrence, written $t' \prec t$ as before (or $(t' : \tau') \prec (t : \tau)$ if we want to indicate the types), each t_i is an immediate subterm occurrence of $op(t_1, \ldots, t_n)$. This means that $M_\tau(t)$ is constructed from the set $\{ M_{\tau'}(t') \mid (t' : \tau') \prec (t : \tau) \}$ in a standard way depending on the operator used to construct t. The definition of $M_\tau(t)$ is commonly referred to as a *semantic clause.*

The relevance of this general definition when we try to formalize what is the essence of syntax-directed, semantically based axiomatics should be clear. For each

syntactic construct *op* and its corresponding semantic clause there should be an inference rule. There should be assertion languages and proof systems for each type τ, so that if we write $(t \models A) : \tau$ to indicate a typical assertion about a term t of type τ, this inference rule typically would take the form:

$$\frac{\{(t' \models A') : \tau' \mid (t' : \tau') \prec (t : \tau)\}}{(t \models A) : \tau}$$

with a premise for each immediate subterm occurrence of t. Although this formulation may look somewhat awkward, it collapses to the usual Hoare rules when applied to while-programs and partial correctness semantics, with the adoption of the usual syntactic sugaring and suppression of syntactic types. And for each type τ it should be possible to define an appropriate notion of implication on assertions about objects of that type, so that the proof system for that type would include a *modus ponens* rule. Of course, the proof systems for the various types fit together in a hierarchical manner which mimics the syntactic structure of the programming language; and the various implication relations at each type may require a mutually recursive definition.

An appropriate generalization of the use of proof outlines in the Cook theorem and its proof would then be the following. If $(t \models A) : \tau$ is a valid assertion (validity being defined by appeal to the semantics of the language, of course), then there is an (type-respecting) assignment *assert* of *assertions* to the (immediate) subterm occurrences of t such that $(t \models A) : \tau$ is deducible from the application of this rule to the premises

$$\{(t' \models \mathit{assert}(t')) : \tau' \mid (t' : \tau') \prec (t : \tau)\}$$

This again collapses to the Cook theorem for while-programs in that simple case. And in the example of parallel programs this seems an appropriate generalization of the relevant theorem.

RELATED WORK AND FURTHER RESEARCH

We have not described a general-purpose technique for constructing semantically based, syntax-directed proof systems. Rather, our as yet limited experience gained while investigating some particular programming languages and program properties has led us to make some (we hope) reasonably coherent guidelines. Our formulation of the denotational setting and its corresponding axiomatic analogue in the previous section is intended as a first step towards a general theory. We believe that the ideas are much more widely applicable than we have been able to indicate here, and a general theory would be very worthwhile. The recent work of Abramsky [1] may turn out to be an important contribution towards such a general theory; he aims at a logical presentation of the domain theoretic constructs prevalent in

denotational semantics. In a similar vein we also mention the recent developments of Robinson, also exploring the axiomatics of denotational semantics ([21]). The full implications of this work need to be worked out, and the connections with existing proof systems could be interesting. One particularly interesting example should be provided by the work of Stirling [22], who has developed a compositional (syntax-directed) formulation of the Owicki-Gries proof system involving a different type of assertion from ours.

There are several interesting issues from an axiomatic point of view which remain to be explored. A fairly simple example occurs in proof systems for dealing with arrays (see [4] for example) where it is common to find an axiom for assignment to an array position which superficially looks as simple as Hoare's axiom for "ordinary" assignment to a variable:

$$\{\,[E \setminus A[E_0]]P\,\}A[E_0]{:=}E\{\,P\,\}.$$

However, the syntactic definition of what it means to substitute an expression for $A[E_0]$ in a condition is somewhat involved. I believe it would be worth investigating an alternative proof system in which a component of the proof system involves reasoning about the (R-)value ([24]) of an (index) expression. It may be useful to allow reasoning about assertions such as "the value of E_0 lies in a certain range", with the obvious intention. One might then design a proof rule of the following form.

$$\frac{\{\,E_0 : X\,\} \quad \bigwedge_{v \in X}([E \setminus A[v]]Q \Rightarrow P)}{\{\,P\,\}A[E_0]{:=}E\{\,Q\,\}}$$

where $E_0 : X$ is an assertion saying that the value of E_0 is in a range described by the set X (a finite subset of the integers). Since the substitution now only involves "simple" array variables $A[v]$ where v is known, it ought to be possible to give a more straightforward syntactic definition. We have ignored issues pertaining to range checking for indices out of bounds. And of course this idea itself brings other problems, such as the possible need for more complex assertions for E_0, and the choice of a syntax for describing finite subsets of integers. Nevertheless, our version of the rule could be argued as more accurately reflecting the semantics and our operational intuitions about what happens in an array assignment. We do not want to get involved here in the details, but we propose to investigate this issue elsewhere.

As we outlined above, syntactic classes other than commands are candidates for syntax-directed reasoning, and an approach that systematically investigates the possibilities may lead to some revision of our current ideas as to the "best" way to reason about programs. Notable work along these lines is reported in [5], where an axiomatization is given for expressions with side-effects.

Much more challenging applications of our ideas are provided by procedural languages, especially when including higher types (procedures as parameters to procedures); the semantic structures necessary to describe partial correctness of programs then become intricate. It will be interesting to see if any benefits can be gleaned from a semantically based approach.

REFERENCES

[1] Abramsky, S., Domain Theory in Logical Form, Proc. Symposium on Logic in Computer Science, Ithaca, NY, IEEE Computer Society Press (1987) 47-53.

[2] Apt, K. R., Ten Years of Hoare's Logic: A Survey, ACM TOPLAS, Vol. 3 (1981) 431-483.

[3] Apt, K. R., Bergstra, J. A., and Meertens, G. L. T., Recursive Assertions are not enough—or are they?, TCS 8 (1979) 73-87.

[4] de Bakker, J. W., Mathematical Theory of Program Correctness, Prentice-Hall (1980).

[5] Boehm, H.-J., Side-effects and Aliasing can have Simple Axiomatic Descriptions, ACM TOPLAS, vol. 7, no. 4 (1985) 637-655.

[6] Brookes, S. D., An Axiomatic Treatment of a Parallel Language, Proc. Symposium on Logics of Programs, Springer LNCS 193 (1985) 41-60.

[7] Brookes, S. D., A Semantically Based Proof System for Deadlock and Partial Correctness in CSP, Proc. Symposium on Logic in Computer Science, IEEE Computer Society Press (1986) 58-65.

[8] Brookes, S. D., A Fully Abstract Semantics and a Proof System for an ALGOL-like Language with Aliasing, Proc. Conference on Mathematical Foundations of Programming Semantics, Manhattan, Kansas, Springer LNCS 239 (1985) 59-100.

[9] Clarke, E. M., The Characterization Problem for Hoare's Logic, in: Mathematical Logic and Programming Languages, eds. C. A. R. Hoare and J. C. Shepherdson, Prentice-Hall (1986) 89-103.

[10] Clarke, E. M., Programming Language Constructs For Which It Is Impossible To Obtain Good Hoare Axiom Systems, JACM Vol. 26 No. 1 (January 1979) 129-147.

[11] Cook, S., Soundness and Completeness of an Axiom System for Program Verification, SIAM J. Comput 7 (1978) 70-90.

[12] Dijkstra, E. W., A Discipline of Programming, Prentice-Hall (1976).

[13] Floyd, R., Assigning Meanings to Programs, in: J. T. Schwartz, ed., Mathematical Aspects of Computer Science, Proc. Symp. Applied Math. (American Math. Soc. Providence) Vol. 19 (1967) 19-32.

[14] Hoare, C. A. R., An Axiomatic Basis for Computer Programming, CACM 12 (1969) 576-580).

[15] Hennessy, M. C. B., and Plotkin, G. D., Full Abstraction for a Simple Parallel Language, Proc. MFCS 1979, Springer LNCS 74 (1979) 108-120.

[16] Milner, R., A Calculus of Communicating Systems, Springer LNCS 92 (1980).

[17] Milner, R., Fully Abstract Models of Typed Lambda-Calculi, Theoretical Computer Science vol. 4 no. 1 (1977) 1-22.

[18] O' Donnell, M., A Critique of the Foundations of Hoare-style Programming Logic, CACM vol. 25 no. 12 (December 1982) 927-934

[19] Owicki, S. S., Axiomatic proof techniques for parallel programming, Ph.D. thesis, Cornell University (1975).

[20] Owicki, S. S., and Gries, D., An Axiomatic Proof Technique for Parallel Programs, Acta Informatica 6 (1976) 319-340.

[21] Robinson, E., Axiomatic Aspects of Denotational Semantics, preprint, Cambridge University (1986).

[22] Stirling, C., A Compositional Reformulation of Owicki-Gries's Partial Correctness Logic for a Concurrent While Language, Proc. ICALP 1986, Springer LNCS 226 (1986) 407-415.

[23] Stoughton, A., Fully Abstract Models of Programming Languages, Ph. D. thesis, Department of Computer Science, Edinburgh University (1986).

[24] Stoy, J., Denotational Semantics, MIT Press (1977).

[25] Strachey, C., The Varieties of Programming Language, Proceedings of International Computing Symposium, Cini Foundation, Venice (1972) 222-233.