Reasoning About Procedures as Parameters in the Language L4

Steven M. German, Edmund M. Clarke, Joseph Y. Halpern

September, 1988

CMU-CS-88-181

Steven M. German; GTE Laboratories, Inc., 40 Sylvan Rd.,
Waltham, MA 02254

Edmund M. Clarke; Dept. of Computer Science, Carnegie-Mellon University,
Pittsburgh, PA 15213

Joseph Y. Halpern; IBM Almaden Research Center, 650 Harry Rd.,
San Jose, CA 95120

This research was sponsored in part by the National Science Foundation under Grant Number CCR-8722633. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation or of the U.S. Government.

Reasoning About Procedures as Parameters in the Language L4

Steven M. German, Edmund M. Clarke, Joseph Y. Halpern

ABSTRACT. We provide a sound and relatively complete axiom system for partial-correctness assertions in an Algol-like language with procedures passed as parameters, but with no global variables (traditionally known as the language L4). The axiom systems allows us to reason syntactically about programs, and construct proofs for assertions about complicated programs from proofs of assertions about their components. Such an axiom system for a language with these features had been sought by a number of researchers, but no previously published solution has been entirely satisfactory. Our axiom system extends the natural style of reasoning used in previous Hoare axiom systems to programs with procedures of higher-type. The details of the proof that our axiom system is relatively complete in the sense of Cook may be of independent interest, because we introduce results about expressiveness for programs with higher types that are useful beyond the immediate problem of the language L4. We also prove a new incompleteness result that applies to our logic and to similar Hoare logics.

Table of Contents

1. Introduction

In order to introduce the new results of this paper, it will be helpful to briefly review some previous work on Hoare axiom systems. It was observed in [Cl79] that there cannot be an axiom system for partial-correctness assertions that is sound and relatively complete in the sense of Cook [Co78] for an Algol-like language with procedures passed as parameters and unrestricted use of global variables. Subsequently, much attention has been given to the problem of axiomatizing Algol without global variables, the language called L4 in [Cl79]. For the first time in the literature, we present an axiom system and show that it is sound ant relatively complete for L4 in the sense of Cook.

Semantically, procedures passed as parameters in L4 are a very powerful feature because the body of a procedure can contain free references to formal procedures. During the execution of an L4 program, this can give rise to chains of procedure references that can grow arbitrarily long.[1] Intuitively, the difficulty in axiomatizing L4 comes from the fact that the chains of procedure references make it possible for a program to reach an unbounded number of distinct procedure environments, or associations between procedure names and bodies. This is known as the infinite range problem. "Conventional" Hoare axiom systems are based on reasoning about either a single procedure environment, or a known, bounded number of environments.

The main new result of this paper is the first real relative completeness proof for L4, with a first-order oracle and the usual notion of expressiveness. Previous attempts to axiomatize languages with higher-type procedures used higher-order assertion languages to make assertions about an unbounded number of procedure environments [Ol81/84, DJ82/83]. This led to completeness relative to higher-order theories, not relative to the first-order theory of the interpretation. Also, it was necessary in these approaches to assume the interpretation was expressive in a certain higher-order sense, i.e. that the strongest postcondition of programs with free higher-order parameters could be expressed.

1.1. Historical Background

A partial correctness assertion (pca) is a formula of the form {U} S {V}, where U and V are first-order formulas, and S is a statement of a programming language. If H is a pca and I is a first-order interpretation, we write $I \models H$ to denote that H is semantically true in I. If we have an axiom system in mind, we write $\Gamma \vdash H$ to denote

[1] An example of this can be found in Section 5.

that H is provable from a set of formulas Γ, which is taken as an assumption. In many cases, we must deal with an interpretation I whose first-order theory, written Th(I), is not effectively axiomatizable. For this reason, Th(I) is often used as an assumption in Hoare axiom systems.

In [Co78], Cook discussed logical properties of an axiom system for pcas. The notion of soundness for Hoare axiom systems is straightforward: an axiom system is said to be sound if for all I and H, Th(I) $\vdash$ H implies I $\models$ H. For completeness, we cannot simply use the converse of this condition, because of the problem of expressiveness. That is, it may be that I $\models$ H holds, but H is not provable in the axiom system from Th(I), because assertions needed at intermediate steps in the proof cannot be expressed by pcas with first-order pre- and post-conditions interpreted in I. An interpretation is said to be expressive for a programming language if for every first-order formula U and program π, there is a first-order formula that interpreted in I expresses the strongest postcondition of π with respect to U.[2]

Intuitively, if an interpretation is expressive, then it is possible to write all of the pcas needed as intermediate assertions in proofs. In [Co78], an axiom system is said to be relatively complete if for all pcas H and expressive interpretations I, I $\models$ H implies Th(I) $\vdash$ H. Using this notion of relative completeness, Cook [Co78] and Gorelick [Go75] showed axiom systems for non-recursive and recursive procedures having simple call-by-name parameters to be sound and relatively complete. It is intuitively clear that it would be undesirable to use a notion of completeness weaker than Cook's.

The early work on reasoning about procedures in Hoare's logic dealt only with *simple* procedure parameters, that is, parameters whose values are individuals in the domain of the interpretation. There are many situations in programming languages where there are more complex parameters, such as procedures and functions. Recently, programming languages have incorporated the notions of polymorphic (or generic) and object-oriented programming, both of which involve treating complex objects as parameters. Programming of modules that can be instantiated with different higher-type parameters contributes to reusability, a major goal of software developers. Exception handling mechanisms that allow a procedure to be passed for exceptions are another context where complex parameters are used. Thus, the problem of treating complex parameters is a key issue arising in many contexts.

Next, we discuss the work of Clarke [Cl79], who considered axiom systems for Algol with procedures passed as parameters. In constrast to the earlier results giving sound

---

[2]The reader will find formal definitions of these familiar terms in Section 3.2.

and relatively complete axiom systems for langauges with simple parameters, it was shown in [Cl79] that for sufficiently complex Algol-like languages there cannot be a Hoare axiom system that is sound and relatively complete. This incompleteness result was quite surprising at the time. The incompleteness exists whenever a programming language contains (or can simulate) the following combination of features: (i) procedures with procedures passed as parameters, (ii) recursion, (iii) use of non-local variables, (iv) static scoping, and (v) local procedure declarations. The full language with these five features was called L1 in [Cl79]. It is also shown in [Cl79] that if any one of the features (i), (ii), (iv), or (v) are dropped from L1 Algol, a sound and relatively complete axiomatization can be obtained for the resulting languages (called L2, L3, L5, and L6 in [Cl79]). It was conjectured that the same is true for the language L4 which results when feature (iii), use of non-local variables, is dropped.

The languages L2, L3, L5, and L6 are relatively easy to axiomatize, since they all have the finite range property. Informally, this property is that for each program, there is a bound on the number of distinct procedure environments that can be reached. Intuitively, procedures passed as parameters are a weak feature in a language with finite range, because only a finite number of distinct procedures can occur as actual parameters in any one program. It is possible to prove properties of programs in a language with finite range by simply treating each of the possible procedure environments as a separate case. Thus the presence of procedures as parameters does not greatly complicate reasoning. The language L4, however, does not have the finite range property. Intuition suggested that some new reasoning methods would be needed for such a language. This intuition was supported by Olderog in [Ol81/83], where a precise characterization was given for the class of Hoare axiom systems based on copy rules, and it was shown that none of these axiom systems can deal adequately with infinite range.

Infinite range is also important in other contexts. For example, to reason about a generic library package that can be instantiated with an infinite number of different procedure parameters, it is necessary to deal with infinite range.

The results of Clarke and the characterization theorem of Olderog had the effect of focusing much research attention on the language L4, which has infinite range. In addition, there was another line of investigation which indicated that sound and relatively complete axioms for L4 might exist (although it did not directly show how to find them!). Clarke's incompleteness result for the full language was based on an observation about the relationship between the halting problem for a programming language in finite interpretations and the existence of a sound and relatively complete

axiom system. The halting problem referred to is: given a program and a finite interpretation, to tell whether the program can halt in the interpretation for some initial assignment of values to its variables. Clarke observed that if the halting problem for a language in finite interpretations is undecidable, then it is not possible to obtain a sound and relatively complete axiom system for partial correctness. The incompleteness result of [Cl79] is established by showing that the halting problem for the full language L1 is undecidable.

In the sequence of papers [Li77], [CGH83], and [Gr84/86], a sort of converse to the incompleteness result was established. It was shown that if the halting problem for an *acceptable programming language* [CGH83] is decidable for finite interpretations, then for an expressive interpretation I, the set of pcas true in I is uniformly recursively enumerable in Th(I). Acceptability of a programming language is a mild technical condition that is easily satisfied by "reasonable" programming languages including Algol and the sublanguage L4. In the case of L4, additionally, it was shown by Langmaack [La82] that the halting problem in finite interpretations is decidable. Hence, one could conclude that for L4, the pcas true in an expressive interpretation are uniformly r.e. in the theory of the interpretation. Note that asserting the existence of a uniform effective procedure for enumerating pcas is quite different from asserting the existence of or exhibiting an axiom system based on the syntax of programs in the style of Hoare. While the results strongly suggested that there could be a sound and relatively complete axiom system for L4, the problem of actually finding such an axiom system and proving its completeness remained open for several years.

Partial solutions were given in [Ol81/84, DJ82/83]. However, these papers established completeness relative to higher-order theories, not relative to the first-order theory of the interpretation. This was unsatisfactory because the oracle required in order to reason about a program in these axiom systems had to be increased to the highest type used in a procedure in the program. In other words, in order to reason about programs over a fixed interpretation, it is necessary to use increasingly larger, higher-order oracles, depending on the types of procedures appearing in the programs. Moreover, it is necessary in these approaches to assume the interpretation is expressive in a certain higher-order sense, i.e. that the strongest postcondition of programs with free higher-order parameters can be expressed.

In [GCH83], we addressed these problems by introducing an axiom system for L4 in which the assertion language is built up from ordinary first-order formulas. We showed the soundness of the axioms, and very briefly described a proof that the axiom system is relatively complete, in Herbrand definable interpretations, relative to the first-order

theory of the interpretation and using only the ordinary assumption of expressiveness. A Herbrand definable interpretation is one in which every value in the domain is given by a variable-free term. The relative completeness applies to programs with arbitrary finite-depth procedure types, not just the Pascal case (depth-one) considered in [Ol81/84].

The higher-order notion of expressiveness used in [DJ82/83] has been shown to be equivalent to the usual notion of [Co78], for Herbrand definable interpretations [Jo83]. More recently, another Hoare calculus that can be used to reason about procedures as parameters has been shown to be relatively complete, relative to the first-order theory of the interpretation and with the usual notion of expressiveness, for Herbrand definable interpretations [Go85]. These results use a characterization [GH83, Ur83] of expressiveness on Herbrand definable interpretations. By restricting attention to Herbrand definable interpretations, the class of expressive interpretations is narrowed in a way that admits certain simple arguments about encoding power. Other relative completeness proofs for Herbrand definable interpretations have appeared in [CGH83, Cl84].

The assumption of Herbrand definability is natural in the sense that most of the interpretations used in practical computing are Herbrand. Nevertheless, Herbrand definability can be criticized as a hypothesis in relative completeness theorems [Gr84/86]. A syntactic Hoare axiom system is intended to capture the meaning of a programming language in a way that is independent of the interpretation. If an axiom system is relatively complete for Herbrand interpretations but not otherwise, it has more dependence on the interpretation than "conventional" axiom systems, such as the familiar axiom system for while-programs.

This paper has been several years in preparation, since our original presentation [GCH83] in 1983. A shorter version of the present paper was presented in 1986 [GCH86a], describing for the first time an approach to obtaining relative completeness without the assumption of Herbrand definability. An unabridged draft containing full details of the proofs was also circulated to interested researchers in the summer of 1986 [GCH86b].

1.2. New results of this paper

The main new result is the first real relative completeness proof for L4, with a first-order oracle and the usual notion of expressiveness. The key to otaining this result is a method of constructing a program that simulates procedures passed as parameters, using only ground variables. A statement is said to be a program if it has no free

procedures. For any L4 program, we construct another L4 program, without procedures as parameters, that simulates the original program in a given interpretation. The simulating program has a syntactic structure that is closely related to the original one, making it possible to use it to prove properties of the original program by structural induction.

The simulation depends on interpreter programs, which are programs that can simulate any procedure of a fixed type. The interpreter programs use an encoding of procedures to control which procedure is simulated. Procedures passed as parameters are represented as closures, or elements of a set of procedure declarations generated from the declarations in the program and having no free procedure identifiers. To construct the simulation, we define four basic operations on closures that are sufficient for dynamically simulating the computation of an L4 program. We then show that closures can be represented by ordinary ground values of an interpretation. In any interpretation having more than one domain value, there are L4 programs (with no procedures as parameters) that can compute approximations to the basic operations we need on closures. (In the case of interpretations with only one value in the domain, relative completeness of the axiom system can be proved in a straightforward way without the simulation.) These programs are used to construct, for any L4 program, the L4 program with no procedures as parameters that simulates the original one.

For a partial-correctness assertion, $\{U\}\ \pi\ \{V\}$, true in an interpretation I, we then prove $\{U\}\ \pi\ \{V\}$ in a syntactic axiom system, by the usual induction on the structure of π. The program that simulates π is only needed to show the existence and define the semantics of certain first-order formulas used in the proof.

The strategy of proving relative completeness of a language with higher-order objects by simulating the language over ground variables appears to be useful beyond the immediate problem of L4. This technique gives useful insights into the semantic effect of various language restrictions, and can help to suggest other languages that may also be axiomatizable. In this paper, we emphasize aspects of the completeness proof that are applicable to other problems in the area.

Although we prove that our axiom system is relatively complete for partial-correctness assertions (pcas), our logic allows more complicated formulas than just pcas. For example, we allow implications between pcas and quantification over ground-type variables (i.e. variables ranging over the domain of the interpretation) in complex formulas built up from pcas. A natural question is whether there can be a sound and relatively complete axiom system for the full logic. We show that this is not possible. More precisely, we show that for any *deterministic* program π, it is possible to

effectively find a formula of our logic that is semantically true in I iff π is totally correct with respect to a set of first-order pre- and postconditions. It is well known that there cannot be an axiom system based on only a first-order oracle that is sound and relatively complete for total-correctness assertions involving, for example, while-programs [Ap81] (see also [Gr85]). Thus there cannot be an axiom system based on only a first-order oracle that is sound and relatively complete for all formulas of our logic. This result appears to apply to other recently proposed Hoare logics that form formulas for higher-order procedures by implication between arbitrary formulas and by quantification over domain values in higher-order formulas, such as [Si85].

The organization of the rest of this paper is as follows: Section 2 gives an informal overview of the completeness proof. Section 3 defines the syntax and semantics of the programming language L4. Section 4 defines the syntax and semantics of formulas in our Hoare logic and describes the axiom system. In Section 5, we illustrate the use of the axiom system with an example. Section 6 presents an analysis of the semantics of L4, which is the first part of the completeness proof. Section 7 gives the part of the completeness proof that depends on the logic given in Section 4. Section 8 shows how to express total correctness in the logic. Section 9 discusses possible extensions beyond L4. In Section 10 we present our conclusions.

2. Overview

Our main result is the following theorem. The first half of the proof, given in Section 6, is independent of the axiom system, which is essentially the same as in [GCH83]. As is usual, we consider the truth of a partial correctness assertion (pca) to be defined with respect to a first-order interpretation, and we write $I \models \{U\}\ \pi\ \{V\}$ to denote that the assertion is semantically true in the interpretation I. We let Th(I) denote the first-order theory of I, and $Th(I) \vdash \{U\}\ \pi\ \{V\}$ means that the assertion is provable in our axiom system by taking Th(I) as assumptions.

Theorem. Let π be a program in L4 and I be an expressive interpretation. Then $I \models \{U\}\ \pi\ \{V\}$ implies $Th(I) \vdash \{U\}\ \pi\ \{V\}$.

The axiom system and the relative completeness proof were developed simultaneously. Initially, it was far from clear what axioms would be needed to give a sound and relatively complete system for L4. Some of the axioms were discovered by considering examples or by trying to construct the completeness proof. On a very informal level, the main idea of the proof is to prove a lemma of the form

$$Th(I) \vdash \forall \bar{q}\ (H(\bar{q}) \rightarrow \{P(\bar{x})\}\ S\ \{SP\,[S;\ P(\bar{x});\ H(\bar{q})]\}),$$

for arbitrary statements S, by structural induction. Intuitively, in L4, since a statement S can be executed in an infinite number of different procedure environments, we must be able to reason about the meaning of S in an infinite number of distinct procedure environments. We have expressed this informally by introducing an assumption $H(\bar{q})$ about the free procedures in S, and an informal notation $SP\,[S;\ P(\bar{x});\ H(\bar{q})]$ for the strongest postcondition of S, given a precondition $P(\bar{x})$ and assumption $H(\bar{q})$ about free procedures. Note that $SP\,[S;\ P(\bar{x});\ H(\bar{q})]$ is a *semantic* concept; we have not yet said how the components of the lemma are to be formalized in a logic.

Our reason for beginning on a deliberately vague level is to point out that there is more than one approach that can be taken to making things precise, and that there are several technical problems that must be overcome.

First of all, because $\bar{q}$ ranges over an infinite number of possible procedures, $SP\,[S;\ P(\bar{x});\ H(\bar{q})]$ must be a formula that ranges over an infinite number of different relations (in some interpretations), depending on $\bar{q}$. This seems to suggest the use of higher-order formulas in partial-correctness assertions, an approach taken in [Ol81/84, DJ82/83]. But, there are two technical problems that are encountered with this approach.

In conventional Hoare-style axiom systems, there is a rule of consequence:

$$\frac{U \supset U1,\ \{U1\}\ S\ \{V1\},\ V1 \supset V}{\{U\}\ S\ \{V\}}$$

This rule is useful when there is an oracle for Th(I), because $U \supset U1$ and $V1 \supset V$ are first-order formulas. However, if partial-correctness assertions can contain higher-order formulas, then we need a higher-order oracle or further analysis to show that the first-order oracle is sufficient.

The second technical problem concerns expressiveness. The familiar notion of expressivness [Co78] is that an interpretation is expressive for a language if for any program π (in Algol, a statement without free procedures), and any first-order precondition U, the strongest postcondition of π with respect to U, SP[π; U], is expressed by a first-order formula. It is not immediately clear that this assumption of expressiveness allows us to express the higher-order SP[S; P; $H(\bar{q})$], where S can have free procedures and P is higher order.

It may be possible to solve these problems. For instance, the higher-order notion of expressiveness has been shown to be equivalent to the ordinary notion, for Herbrand definable interpretations [Jo83]. However, the proof in [Jo83] depends on the characterization of expressiveness in Herbrand definable interpretations [GH83, Ur83]. Some of the new techniques in our relative completeness proof may be applicable toward removing assumptions of Herbrand definability in other axiom systems.

In contrast to the approach of allowing higher-order formulas in partial-correctness assertions, we allow only first order formulas. This preserves rule of consequence. In order to construct partial correctness assertions in which pre- and postconditions depend on the procedure environment, we introduce a new kind of ground variable, called an *environment variable.* Environment variables may appear in formulas in pcas, but not in programs.

With environment variables, the general lemma can have the form

$$\mathrm{Th(I)} \vdash \forall\bar{q}\ \forall\bar{v}\ (H(\bar{q}, \bar{v}) \rightarrow \{U(\bar{x})\}\ S\ \{SP_{S,U,H}(\bar{x}, \bar{v})\}),$$

where $\bar{v}$ is a sequence of environment variables, and $SP_{S,U,H}(\bar{x}, \bar{v})$ is some first-order formula. (Note: SP[S; U] denotes a set of states that may or may not be expressed by a formula; $SP_{S,U,H}(\bar{x}, \bar{v})$, on the other hand, is a first-order formula that depends on S, U, and H.) In our axiom system, the *procedure variables*, $\bar{q}$, can appear only in programs. Semantically, a procedure variable q ranges over a set of syntactic objects, all L4 declarations of the procedure name q. An environment variable ranges over dom(I), and like a procedure variable, it has the same meaning everywhere in its scope, for instance, on both sides of the arrow in the above formula. (The syntax and

semantics of formulas are defined formally in Section 4.1.)

The assumption $H(\bar{q}, \bar{v})$ will be used to express some relationship between the procedure environment and ground values. Then in the pca $\{U(\bar{x})\}\ S\ \{SP_{S,U,H}(\bar{x}, \bar{v})\}$, we must find a first-order formula $SP_{S,U,H}(\bar{x}, \bar{v})$ that expresses, for each $\bar{v}$, the strongest postcondition of S and $U(\bar{x})$, provided $\bar{q}$ is a vector of procedures satisfying $H(\bar{q}, \bar{v})$.

In order to define the first-order formula $SP_{S,U,H}(\bar{x}, \bar{v})$ and the formula $H(\bar{q}, \bar{v})$ that will allow the inductive proof to go through, we analyze the computing power of L4 programs on *bounded* and *unbounded* interpretations, using some of the techniques of [Li77, CGH83]. An interpretation I is said to be bounded for a programming language $\mathcal{P}$ iff for each program $\pi \in \mathcal{P}$ there is a bound n such that for all initial valuations σ, when π is started in state σ, it can reach less than n distinct valuations. An interpretation that is not bounded for $\mathcal{P}$ is said to be unbounded for $\mathcal{P}$.

We show how to simulate an L4 program by using a certain set of declarations, called *closures*, to stand for the values of formal procedures. The closures are declarations having neither free ground variables nor free procedure identifiers. Intuitively, a closure is an object that represents a procedure and its complete environment of free identifiers. In L4, procedures do not have free ground variables, so a closure needs to represent just the procedure and its procedure environment.

Our first step is to analyze the semantics of procedures in the language L4. We define a family of relations, called *simulation relations*, that can be used to simulate procedures passed as parameters using natural numbers to stand for procedures. These relations are:

1. an *encoding relation*, ρ, which assigns a natural number (a code number) to a declaration;

2. a *binding relation*, β, which takes a code number for a main declaration d, and code numbers for an environment E of other declarations, and gives the code for E|d, which is the main declaration with its free procedures fixed to the procedures in E.

3. a *renaming relation*, ν, which takes a code number for a declaration and a new procedure name, and produces a code for a declaration that is semantically equivalent except that it declares a procedure with a different name.

4. an *interpreter relation*, ι, which takes a code number for a declaration, and gives the semantics in the interpretation I of a call on the encoded declaration.

We use different encodings of closures as natural numbers, depending on whether the interpretation is bounded. In an unbounded interpretation, where programs have essentially all the power of arithmetic, the encoding we use is simply a Godel numbering of the declarations. In a bounded interpretation we must be more careful, and the encoding uses only a finite initial segment of the natural numbers. To construct this encoding, we first define the notion of *semantic equivalence of declarations in an interpretation*, for two closed L4 declarations of the same type, i.e. declarations having no free ground variables or procedure names. We define d_1 to be equivalent to d_2 in I iff all calls to d_1 and d_2 with the same higher-type parameters have the same semantics in I on the ground parameters. In a bounded interpretation I, we show that the relation of semantic equivalence in I partitions the closures generated from the declarations of an L4 program into a finite set of equivalence classes. This is a consequence of the fact that L4 declarations do not have global variables. Thus we can assign a unique number to each equivalence class of closures, and we need only a finite number of natural number codes.

By using the encodings and using the power of programs to simulate arithmetic in bounded and unbounded interpretations, we can show that in any interpretation I, $|I|>1$, there are L4 programs that simulate the operations on closures in a sense that is defined precisely in Part 1 of the proof.

From these programs we can construct, for any statement S of a given L4 program, a statement S* *without any procedures as parameters*, having the following properties: (1) The free ground variables of S* are the free ground variables of S, together with some new ground variables that represent encodings of the environment; (2) S* has no global variables in procedures, so that it is in L4; (3) When S* is started in an initial state such that its code variables are set to an encoding of an environment E, it transforms the values of the original ground variables in essentially the same way that S does when run in environment E.

This technique allows us to solve the problem of expressiveness in the main lemma of the relative completeness proof. The ordinary notion of expressiveness would not guarantee that SP[S; P] could be expressed by a first-order formula when S has free procedures. But SP[S*; P] can always be so expressed, because S* is an L4 program.

In the second part of the proof we use the operations on closures and the S* construction to define the formulas mentioned earlier in the main lemma, $H(\bar{q}, \bar{v})$ and $SP_{S,U,H}(\bar{x}, \bar{v})$, and to prove the main lemma by structural induction.

Note that the encodings used in this proof of completeness are not part of the

axiom system in any sense. They are used here because we need a uniform way of constructing certain first-order formulas. We believe that if one is trying to prove a pca involving a well-understood program and the interpretation is not "badly-behaved," it should be possible to find ways of expressing the necessary intermdeiate assertions without resorting to encodings. In fact, we have used the axiom system to prove partial correctness of some nontrivial L4 programs in a very natural way, without using an encoding at all. In Section 5, we give an example that shows that our logic is a very natural one to use for specifying and reasoning about higher-order procedures.

3. The Syntax and Semantics of L4

3.1. Syntax of L4

We will now describe the programming language L4.

To define the set of programs and the formulas used in our axiom system, we begin by fixing a first-order type Σ which determines the finite set of constant, predicate, and function symbols that can appear in programs and first-order formulas.

Ground variables are variables ranging over the domain of the interpretation.

A procedure type is a type for a procedure name. Procedure types are syntactic sequences of the form $\tau = (\tau_1,...,\tau_n,$ var,...,var), $n \geq 0$, that specify the types of the formal parameters of procedures. Specifically, the set of procedure types is defined inductively as follows: A procedure type is either the empty sequence, (), or a sequence of zero or more procedure types followed by zero or more occurrences of the symbol var. In the procedure type τ above, τ_i is a procedure type for the i^{th} formal parameter. The var elements are for ground parameters of procedures.

There are two kinds of procedure names (also called procedure identifiers), declared procedure names, and formal procedure names. We take p to be a typical declared procedure name, and r to be a typical formal procedure name. In the programs that we reason about, declared procedure identifiers are assigned fixed statements (bodies), while formal procedure identifiers are bound variables that take on different meanings at different points in the execution of a program. (Cf. the discussion of declarations below.)

An arbitrary procedure name is an identifier that indicates an occurrence of a procedure name that is either declared or formal; that is, it is a meta-syntactic symbol standing for a procedure name. Arbitrary procedure names are not part of the language, but are simply a notation we use for talking about the language. We take q to be a typical arbitrary procedure name.

Each procedure name has a fixed procedure type.

A statement S has one of the forms:

x := e | S1; S2 | If b Then S1 Else S2 | S1 Or S2 |
Begin var x; S End | Begin E; S End | $p(\bar{q}, \bar{x})$ | $r(\bar{q}, \bar{x})$.

In the statements x :=e and Begin Var x; S End, x is a ground variable. The statement S1 Or S2 makes a nondeterministic choice and executes one of the statements. In Begin E; S End, E is a set of procedure declarations. We often write E|S as an

abbreviation for Begin E; S End. In the statements $p(\bar{q}, \bar{x})$ and $r(\bar{q}, \bar{x})$, p is a declared procedure name, r is a formal procedure name, $\bar{q}$ is a list of arbitrary (either declared or formal) procedure names, and $\bar{x}$ is a list of ground variables. The leftmost procedure name in a procedure call (p and r in the above calls) is referred to as the main procedure name of the call. The parameters in a call must match the type of the main procedure name. That is, if the main procedure name has type $(\tau_1,...,\tau_m, var^n)$, then there must be m procedure name parameters, the i^{th} procedure name must have type τ_i, and there must be n ground parameters in the call.

A declaration of procedure q has the form $q(\bar{r}, \bar{x}) \leftarrow$ statement.

A set of procedure declarations, also called an environment, has the form

$$q_1(\bar{r}_1, \bar{x}_1) \leftarrow \text{statement}_1; \ldots; q_m(\bar{r}_m, \bar{x}_m) \leftarrow \text{statement}_m;$$

and introduces mutually recursive declarations of $q_1,...,q_m$. In the list $q_1,...,q_m$, no identifier may appear more than once. The $\bar{r}_i$ are lists of formal procedure names, and the $\bar{x}_i$ are lists of ground variables. The list $\bar{r}_i, \bar{x}_i$, is called the formal list of q_i. The identifiers in the formal list are called the formal parameters of the declaration. The formal list of a procedure may not contain a procedure name or a ground variable appearing more than once. Note that all of the formal procedure identifiers must precede the formal ground variables in a formal list. The main procedure identifier of a declaration is the identifier to the left of the formal list. We will sometimes write q:d to stand for a declaration d with main procedure identifier q. The type of a declaration is simply the type of the main procedure identifier.

The main procedure identifier in a declaration can be either a declared or a formal procedure identifier. Declarations having a formal procedure identifier as the main procedure identifier are only used for technical purposes in the completeness proof, and they never appear in the programs we reason about in the axiom system. In statements appearing in pcas, all declarations must have declared procedure identifiers for their main procedure identifiers.

An occurrence of an identifier in a statement may be either free or bound in the usual sense (the language is lexically scoped). Note that we allow free procedure identifiers to appear in statements. A program is a statement with no free procedure indentifiers.

A declaration is said to be closed iff it has no free procedure identifiers; i.e. any procedure identifiers free in the body are either formal parameters or the main procedure identifier. A declaration is said to be open iff it is not closed.

A declaration $p(\bar{r}, \bar{x}) \leftarrow B$ is said to have no global variables if all the free ground variables of B are in $\bar{x}$. An environment (statement, program) has no global variables if all its declarations have no global variables. Note that such an environment (statement) *may* have free procedures.

We are primarily concerned with programs that have no global variables. For historical reasons [Cl79], this language is often called L4. In L4, the only ground variables that can be accessed or changed by a procedure call are the actual ground variable parameters in the call. This property helps us to get a sound and relatively complete axiom system for L4.

3.2. Semantics of L4

In this section, we review a standard treatment of the semantics of programs as transformations from valuations to valuations, using copy rules. Let I be an interpretation of the type Σ.

A valuation is a mapping from ground variables to values of the domain of the interpretation. If σ is a valuation then $\sigma(x)$ denotes the value of the ground variable x in the valuation σ. We extend this definition to terms in the usual way: if e is a term in the type Σ then $\sigma(e)$ denotes the value of e in valuation σ. If σ is a valuation and u is a domain value then $\sigma[x \leftarrow u]$ is a valuation that is the same as σ except that the value of x in $\sigma[x \leftarrow u]$ is u. If I is an interpretation then val_I denotes the set of valuations mapping ground variables into values in dom(I), the domain of the interpretation.

If σ is a valuation and Q is a first order formula then $I,\sigma \models Q$ means Q is satisfied in interpretation I by the valuation σ. We will write $\sigma \models Q$ to mean $I,\sigma \models Q$, when the interpretation is clear from the context.

The semantics of a program π in the interpretation I is $M_I(\pi) \subseteq val_I \times val_I$. Intuitively, if $(\sigma, \sigma') \in M_I(\pi)$, then when the program π is started in valuation σ, it can halt in valuation σ'. Similarly, if E is an environment and S is a statement then $M_{I,E}(S)$ is the transformation from valuations to valuations of S in environment E. We take the semantics to be formally defined by copy rules.

We define $M_{I,E}$ first for statements without procedure calls or procedure declarations by induction on the structure of the statement. For statements without procedures, the definition of $M_{I,E}$ is independent of the environment E. The environment E only affects the semantics of statements with free procedures, which we will define later. In this definition, we make use of a new basic statement "error," which diverges in all valuations.

$\mathcal{M}_{I,E}(\text{error}) = \emptyset$

$\mathcal{M}_{I,E}(x := e) = \{(\sigma,\sigma[x \leftarrow \sigma(e)]) \mid \sigma \text{ is a valuation}\}$

$\mathcal{M}_{I,E}(S1; S2) = \{(\sigma,\sigma'') \mid \exists\sigma' \ ((\sigma,\sigma') \in \mathcal{M}_{I,E}(S1) \text{ and } (\sigma',\sigma'') \in \mathcal{M}_{I,E}(S2))\}$

$\mathcal{M}_{I,E}(S1 \text{ Or } S2) = \mathcal{M}_{I,E}(S1) \cup \mathcal{M}_{I,E}(S2)$

$\mathcal{M}_{I,E}(\text{If b Then S1 Else S2}) =$
$\{(\sigma,\sigma') \in \mathcal{M}_{I,E}(S1) \mid I,\sigma \models b\} \cup \{(\sigma,\sigma') \in \mathcal{M}_{I,E}(S2) \mid I,\sigma \models \neg b\}$

$\mathcal{M}_{I,E}(\text{Begin var x; S End}) =$
$\{(\sigma,\sigma') \mid \exists(\delta,\delta') \in \mathcal{M}_{I,E}(S), \delta = \sigma[x \leftarrow \underline{a}], \sigma' = \delta'[x \leftarrow \sigma(x)]\}$
(where $\underline{a}$ is a fixed value in dom(I))

We give meaning to statements with procedure declarations and procedure calls by first converting them to statements without procedure declarations and calls by using an auxiliary function Approx_E^k. Informally, Approx_E^k gives the k^{th} approximation to the fixed-point meaning of a recursively defined procedure in the procedure environment E. We define Approx_E^k by induction on k and the structure of statements.

Substitutions. If x and y are variables, then [x/y] is a substitution of x for y. Substitutions separated by "," are simultaneous. For instance, $[x_1/y_1, x_2/y_2]$ is a simultaneous substitution of x_1,x_2 for y_1,y_2. Similarly, if $\bar{x}$, $\bar{y}$ are lists (of the same size) of variables, all distinct, then $[\bar{x}/\bar{y}]$ is a substition that replaces each variable in the list $\bar{y}$ with the corresponding variable in $\bar{x}$.

1. $\text{Approx}_E^k(\text{error}) = \text{error}$

2. $\text{Approx}_E^k(x := e) = x := e$

3. $\text{Approx}_E^k(S1; S2) = \text{Approx}_E^k(S1); \text{Approx}_E^k(S2)$

4. $\text{Approx}_E^k(S1 \text{ Or } S2) = \text{Approx}_E^k(S1) \text{ Or } \text{Approx}_E^k(S2)$

5. $\text{Approx}_E^k(\text{If b Then S1 Else S2}) = \text{If b Then } \text{Approx}_E^k(S1) \text{ Else } \text{Approx}_E^k(S2)$

6. $\text{Approx}_E^k(\text{Begin var x; S End}) = \text{Begin var x; } \text{Approx}_E^k(S) \text{ End}$
renaming the bound variable x if it appears free in E (see below)

7. $\text{Approx}_E^k(\text{Begin E'; S End}) = \text{Approx}_{E \cup E'}^k(S)$
renaming bound variables in E' if necessary (see below)

8. $\text{Approx}_E^k(q0(\bar{q}, \bar{x})) =$ (see explanation below)

$$\begin{cases} \text{error if } k=0 \text{ and } q0 \text{ is declared in } E \\ \\ \text{Approx}_E^{k-1}(B[\bar{q}/\bar{q}', \bar{x}/\bar{x}']) \\ \qquad \text{if } k > 0 \text{ and the declaration } q0(\bar{q}', \bar{x}') \leftarrow B \in E \\ \\ \text{otherwise } E_k|q0(\bar{q}, \bar{x})[p_{1,k}/p_1, \ldots, p_{n,k}/p_n], \text{ where } E_k \text{ is defined below.} \end{cases}$$

In clause 6 if the bound variable x appears free in E, then we have to rename the x to some fresh variable x′ to avoid capturing the free variable in E. Thus we would get

Begin var x′; $\text{Approx}_E^k(S[x'/x])$ End.

Similarly, in clause 7, if some procedure identifier declared in E′ already appears in E, we have to rename the identifiers in E′ (and all their bound occurrences in S) to avoid naming conflicts.

Clause 8, defining Approx_E^k for a procedure call, requires some explanation. The clause has three cases. The first case applies if the call is to a procedure declared in E, but k=0. In this case, we do not expand the procedure call, and leave an error statement, which diverges. In the second case, where $k > 0$ and the call is to a procedure declared in E, we expand the body of the procedure. The third case is that q0 is not declared in E and $k > 0$. The first two cases are sufficient, without the third case, to define the semantics of statements; we define $M_{I,E}(S)$ below in terms of the approximations of a *program*, even if S has free procedures. Thus in expanding the definition of $M_{I,E}(S)$, we never reach an expansion of $\text{Approx}_E^k(q0(\bar{q}, \bar{x}))$ where q0 is not defined in E; i.e. the details of the third case have no effect on our semantics.

The third case of clause 8 is included only for technical reasons; in the proof of the soundness of the axiom system we will use Approx in a more general way such that the third case can occur. Intuitively, in this case, $\text{Approx}_E^k(q0(\bar{q}, \bar{x}))$ is defined to leave the main procedure q0 unexpanded, giving a call of the form $q0(\bar{q}', \bar{x})$, where each procedure in $\bar{q}'$ is the k^{th} approximation of the corresponding procedure in $\bar{q}$.

To handle the third case, we define, for each environment E, a sequence of environments $E_0, E_1, E_2, \ldots$, which give successive approximations to the environment E. For notational convenience, we introduce a sequence of procedure names to correspond to the successive approximations of each procedure. Thus, for each procedure name p_i in E, we let $p_{i,0}$ be the undefined procedure, $p_{i,1}$ be the next approximation, etc. If E consists of the declarations $p_i(\bar{r}_i, \bar{x}_i) \leftarrow B_i$, i=1,...,n, then E_k

is defined inductively as follows:

$$E_0 = \{p_{i,0}(\bar{r}_i, \bar{x}_i) \leftarrow \text{error} \mid i=1,...,n\}$$

$$E_{k+1} = \{p_{i,k+1}(\bar{r}_i, \bar{x}_i) \leftarrow B_i[p_{1,k}/p_1, ..., p_{n,k}/p_n]\} \cup E_k$$

These definitions will only be used in the soundness proof of Section 4.2.5.

Note that if S is a program then $\text{Approx}^k_E(S) = \text{Approx}^k_\emptyset(S)$ (i.e. $\text{Approx}^k_E(S)$ is independent of E), and $\text{Approx}^k_\emptyset(S)$ does not contain any procedure declarations or procedure calls.

Given a procedure environment E and a statement S, let $E^S = E \cup \{p(\bar{r}, \bar{x}) \leftarrow \text{error} \mid p \text{ appears free in } E|S\}$. Note $E^S|S$ is a program, since it has no free procedure identifiers. To complete our semantics, we define, for any statement S,

$$\mathcal{M}_{I,E}(S) = \cup_k \mathcal{M}_{I,\emptyset}(\text{Approx}^k_\emptyset(E^S|S)).$$

Finally, for a program π, we define $\mathcal{M}_I(\pi)$ to be $\mathcal{M}_{I,\emptyset}(\pi)$.

For a program π and a first-order formula Q, the strongest postcondition of π with respect to Q (in the interpretation I), $SP[\pi;\ Q]$, is defined to be $\{\sigma' \mid \sigma'$ is a valuation and for some valuation σ we have $\sigma \models Q$ and $(\sigma, \sigma') \in \mathcal{M}_I(\pi)\}$.

An interpretation I of a signature Σ is said to be expressive for a programming language $\mathcal{P}$ iff for each program $\pi \in \mathcal{P}$ and first order formula Q of type Σ, there is a first order formula SP such that $\sigma \models SP$ iff $\sigma \in SP[\pi;\ Q]$. In an expressive interpretation, we will write $SP[\pi;\ Q]$ to stand for a first order formula that expresses the strongest postcondition of π with respect to Q.

An interpretation I is said to be bounded for a programming language $\mathcal{P}$ iff for each program $\pi \in \mathcal{P}$ there is a bound n such that for all initial valuations σ, when π is started in state σ, it can reach less than n distinct valuations. An interpretation that is not bounded for $\mathcal{P}$ is said to be unbounded for $\mathcal{P}$.

It is shown in [GH83, Ur83] that in programming languages with recursive procedures (as in L4), there are programs that generate all of the domain elements that are reachable from the initial valuation of the variables.[3] It follows that if an interpretation I is bounded for L4, then $\forall k\ \exists n$ such that given k values, $x_1,...,x_k$, in dom(I), less than n values of dom(I) can be generated from $x_1,...,x_k$ using the constant

[3]Recall that we assume throughout that interpretations give meaning to only a finite number of symbols.

and function symbols. We will use this property of bounded interpretations in the completeness proof.

3.3. Properties of Strongest Postconditions

In this section, we list some general properties of strongest postconditions that will be needed in the proof of the Completeness Theorem. The properties that we use could be formally derived from the operational semantics of programs given in section 3.2. However, since we only take strongest postconditions of *programs*, the reader should be able to verify that these properties hold for any semantics that assigns the "standard" meaning to programs as relations on valuations. Many similar properties of strongest postconditions of programs have been used in previous relative completeness proofs, for example [Co78, Cl79].

In the following, π, $\pi 1$, and $\pi 2$ are arbitrary programs. We will say that a program does not change a variable if the semantics of the program does not contain a pair of valuations that assign different values to the variable. Some of the properties of strongest postconditions are true only for programs that do not change certain variables.[4]

The following properties may be read in two ways. In an expressive interpretation, then each property can be read as asserting that two first-order formulas are equivalent in the interpretation. The left and right formulas in each property can also be read semantically, as relations on valuations. In this case, the logical symbols should be read as operations on relations, in the obvious way (for example, $f_1 \wedge f_2$ is the intersection of the relations given by f_1 and f_2.)

SP 0. $SP[\pi 1;\ Q] \equiv SP[\pi 2;\ Q]$
provided $\mathcal{M}_I(\pi 1) \equiv \mathcal{M}_I(\pi 2)$.

SP 1. $SP[\pi;\ Q1] \wedge Q2 \equiv SP[\pi;\ Q1 \wedge Q2]$
provided no variables free in Q2 are changed by π.

SP 2. $SP[\pi;\ Q][x/y] \equiv SP[\pi[x/y];\ Q[x/y]]$
provided x is not free in π or Q.

SP 3. $\exists x\ SP[\pi;\ Q] \equiv SP[\pi;\ \exists x\ Q]$
provided x is not free in π.

---

[4]Of course, one cannot effectively tell in general whether a program changes a variable, but in the proof we will only use properties of strongest postconditions in situations where it is clear that the necessary assumptions are satisfied.

SP 4. $SP[(\pi 1;\ \pi 2);\ Q] \equiv SP[\pi 2;\ SP[\pi 1;\ Q]]$

SP 5. $SP[(\text{Begin var } x;\ \pi \text{ End});\ Q] \equiv \exists x\ SP[\pi;\ Q]$
provided x is not free in Q.

SP 6. $SP[(\pi 1 \text{ Or } \pi 2);\ Q] \equiv (SP[\pi 1;\ Q] \vee SP[\pi 2;\ Q])$

SP 7. $SP[(\text{If b Then } \pi 1 \text{ Else } \pi 2);\ Q] \equiv (SP[\pi 1;\ b \wedge Q] \vee SP[\pi 2;\ \neg b \wedge Q])$

4. The Logic

4.1. Syntax and Semantics of Formulas

Recall that in Section 3.1, we fixed a first-order type Σ which determines the finite set of constant, predicate, and function symbols that can appear in programs and first-order formulas.

We permit three distinct kinds of variables in the logic. There are two kinds of variables ranging over individuals in the domain of the interpretation: ordinary variables and environment variables. The syntactic distinction between ordinary and environment variables is that ordinary variables, like the variables in most Hoare axiom systems, may appear in both programming language statements and first-order formulas; environment variables are a new class of variables which may appear only in first-order formulas. All of the ground variables of a programming language statement are considered ordinary variables of the logic when the statement appears in a pca in the logic.

Finally, there are procedure variables, which may appear in pcas only in the statement part, and can be universally quantified in formulas. All of the procedure identifiers of a statement are considered procedure variables of the logic when the statement appears in a pca in the logic.

Subject to these restrictions on the use of variables, a formula has the form

$$\text{<formula>} ::= U \mid \{U\}\ S\ \{V\} \mid (H1 \wedge H2) \mid (H1 \rightarrow H2) \mid \forall v\ H \mid \forall q\ H$$

where U and V are first-order, S is any statement, H, H1, and H2 are formulas, v is an environment variable, and q is a procedure variable.

Arbitrary nesting of $(H1 \wedge H2)$, $(H1 \rightarrow H2)$, $\forall v\ H$, and $\forall q\ H$, is permitted. As usual, we view $H1 \leftrightarrow H2$ as an abbreviation for $(H1 \rightarrow H2) \wedge (H2 \rightarrow H1)$, and $\neg H$ as an abbreviation for $H \rightarrow$ False.

The semantics of a formula is the set of assignments to the environment and procedure variables that satisfy the formula. This is defined formally below. In terms of the semantics, formulas never have free ordinary variables. If a first-order formula appears alone as a formula of the logic, the semantics is that any free ordinary variables are implicitly universally quantified. Similarly, in a pca, all of the ordinary variables are implicitly universally quantified.

The formulas of our logic form a many-sorted first-order language built up from the pure first-order formulas of type Σ and pcas. The formula $H1 \wedge H2$ is semantically the

conjunction of H1 and H2; H1 $\rightarrow$ H2 is semantically a first-order implication; and quantification in $\forall$v H and $\forall$q H has the usual first-order semantics.

In order to give meaning to formulas we need an interpretation I, which gives meaning to the symbols in Σ in the usual way, an environment valuation ξ which assigns an element of dom(I) to each environment variable, a state σ which assigns an element of dom(I) to each ordinary variable, and a procedure environment E.

$I,E,\xi \models U$ iff for all σ, we have $I,\xi,\sigma \models U$
(where this is defined in the usual way).

$I,E,\xi \models \{U\}\ S\ \{V\}$ iff
for all σ, σ' we have $(I,\xi,\sigma \models U$ and $(\sigma,\sigma') \in \mathcal{M}_{I,E}(S))$ implies $I,\xi,\sigma' \models V$.

$I,E,\xi \models H1 \wedge H2$ iff $I,E,\xi \models Hi$, i=1,2.

$I,E,\xi \models H1 \rightarrow H2$ iff $I,E,\xi \models H1$ implies $I,E,\xi \models H2$.

$I,E,\xi \models \forall v\ H$ iff for all $d \in$ dom(I) we have $I,E,\xi[v \leftarrow d] \models H$.

$I,E,\xi \models \forall q\ H$ iff for all L4 procedure declarations $q'(\bar{r}, \bar{x}) \leftarrow B$ we have
$I, E \cup \{q'(\bar{r}, \bar{x}) \leftarrow B\}, \xi \models H[q'/q]$,
where q' is a fresh variable which does not appear in E and has the same type as q.

$I \models H$ iff for all L4 environments E and for all ξ we have $I,E,\xi \models H$.

The place where our logic differs from standard first-order semantics is in the semantics of pcas. Ordinary variables in pcas have a special meaning and are "bound" variables in the sense that the semantics of a pca involves universal quantification over valuations.

Note that the meaning of a free environment variable in a formula is the same wherever it appears. In contrast, the meaning of an ordinary variable is "local" to each partial correctness assertion in which it appears, since it is effectively universally quantified. For example, consider the following two formulas

(1) {True} y : = y {x = 3} $\rightarrow$ {True} y : = y {False}

(2) {True} y : = y {v = 3} $\rightarrow$ {True} y : = y {False}

where x and y are ordinary variables and v is an environment variable. Formula 1 is valid, because the antecedent {True} y : = y {x = 3} is false: it is not the case that

for all initial values of x and y, y : = y sets x to 3. Formula 2 is not valid (in all interpretations with more than one domain element), because v is quantified over the whole formula. For the value v = 3, the antecedent is true but the consequent is false, giving a counterexample to (2).

4.2. Axiom System

As we discussed, the formulas H1 $\wedge$ H2, H1 $\rightarrow$ H2, $\forall$v H, and $\forall$q H have first-order semantics (in which the ordinary variables in pcas are regarded as bound variables). To reason about these formulas, the axiom system contains a standard deductive system for many-sorted first-order logic [En72], for formulas built up from environment and procedure variables, conjunction H1 $\wedge$ H2, implication H1 $\rightarrow$ H2, negation, and universal quantification $\forall$v H, $\forall$q H. This deductive system regards the pcas as atomic formulas with procedure and free environment variables (only). Ordinary variables are always "bound" for the purposes of this deductive system. Since we are using standard first-order reasoning, the same deductions will follow from any complete deductive system for the first-order predicate calculus.

To make this precise, we will now define the notion of when a formula H is valid under first-order semantics. We will take all such formulas as axioms in the axiom system. First, we will define a new first-order language which will have, in addition to the symbols of Σ, a new predicate symbol for each partial correctness assertion. For each pca H = {U} S {V}, we introduce a new predicate symbol $P_H(\bar{v}, \bar{q})$, which we will call a pca predicate symbol. The pca predicate symbol P_H has one parameter of ground type for each environment variable free in H, and one parameter of a procedure type for each procedure name free in H.

1. Let a primary formula be either a first-order formula or a pca predicate $P_H(\bar{t}, \bar{q})$.

2. Let a wff (well formed formula) have the form

<wff> ::= <primary formula> | wff1 $\wedge$ wff2 | wff1 $\rightarrow$ wff2 | $\forall$v wff | $\forall$q wff.

Intuitively, a wff is like a formula of our logic, except that instead of partial correctness assertions, there are atomic formulas whose free variables are the free variables that can appear in pcas.

Then we say that a wff w is valid if w′, the result of replacing all occurrences of $\rightarrow$ in w by $\supset$, is a valid first-order formula (i.e. true in all interpretations). Let us define the expansion of a wff w to be the formula of our logic obtained from w by replacing every primary formula of the form $P_H(\bar{t}', \bar{q}')$ by the pca $(\{U\}\ S\ \{V\})[\bar{t}'/\bar{v}, \bar{q}'/\bar{q}]$, where H is the pca {U} S {V}, $\bar{v}$ is the list of free environment variables in U and V, and $\bar{q}$ is

the list of free procedure names in S. The lists $\bar{v}$ and $\bar{q}$ are given in the order of the first free appearance of each variable in the text {U} S {V}. Finally, we say that a formula H is valid under first-order semantics if it is the expansion of a valid wff. We take all formulas that are valid under first-order semantics as axioms.

The axiom system also contains standard Hoare axioms for constructs such as assignment and conditional and for conventional reasoning about pcas. The unconventional element of the axiom system is the Recursion Rule R4, which is used for reasoning about higher-order procedures.

We formulate the axiom system to be independent of the particular interpretation. To prove that a formula holds in an interpretation I, one would prove the formula by using the first-order theory of I as an assumption.

4.2.1. Axiom Schemes

Notation. We adopt the following notational convention: If E is an environment and H is a formula, then E|H is the result of replacing every pca {U} S {V} in H by {U} E|S {V}, subject to the usual conditions about renaming variables bound by universal quantifiers, to avoid capture of free variables in E.

Ax 0. H, provided H is valid under first-order semantics.

Ax 1. {True} S {True}

Ax 2. {U[e/x]} x := e {U}

Ax 3. (({U} S1 {V}) $\wedge$ ({V} S2 {W})) $\rightarrow$ {U} S1; S2 {W}

Ax 4. (({U $\wedge$ b} S1 {V}) $\wedge$ ({U $\wedge$ $\neg$b} S2 {V})) $\rightarrow$ {U} If b Then S1 Else S2 {V}

Ax 5. (({U} S1 {V}) $\wedge$ ({U} S2 {V})) $\rightarrow$ {U} S1 Or S2 {V}

Ax 6. {U} S[x′/x] {V} $\rightarrow$ {U} Begin var x; S End {V},
where x′ does not appear in U, V, or S.

Ax 7. ($\forall\bar{x}$(U1 $\supset$ U) $\wedge$ ({U} S {V}) $\wedge$ $\forall\bar{x}$(V $\supset$ V1)) $\rightarrow$ {U1} S {V1},
where $\bar{x}$ is the list of ordinary variables free in U, U1, V, and V1.

Ax 8. {U} S {V} $\rightarrow$ {$\exists$x U} S {$\exists$x V}, if x is an ordinary variable not free in S.

Ax 9. {U} S {V} $\rightarrow$ {U $\wedge$ Q} S {V $\wedge$ Q}, if no variable free in Q is also free in S.

Ax 10. $E|H \rightarrow H$, provided none of the procedures bound in E are free in H.

Ax 11. $\{U\}\ S\ \{V\} \rightarrow \{U\theta\}\ S\theta\ \{V\theta\}$, where θ is an injective function mapping ordinary variables to ordinary variables.[5]

Ax 12. $\{U\}\ S\ \{V\} \leftrightarrow \{U\}\ S'\ \{V\}$, where S' is the result of renaming bound variables (declared procedure names, names in the formal lists of procedures, and local variable names) in S. As usual, such a renaming must not introduce a conflict by making distinct names the same.

4.2.2. Rules of Inference

R0. $$\frac{H1,\ H1 \rightarrow H2}{H2}$$

R1. $$\frac{H}{\forall v\ H,\ \forall q\ H}$$

where v is an environment variable and q is a procedure name.

R2. $$\frac{H \rightarrow H1}{H \rightarrow E|H1}$$

provided no procedure declared in E is free in H.

R3. $$\frac{H \rightarrow \{U\}\ S\ \{V\}}{H \rightarrow \{\exists v\ U\}\ S\ \{\exists v\ V\}}$$

provided v is not free in H.

---

[5] See section 4.3 for a discussion of aliasing in procedure calls.

R4. Recursion Rule. Our Recursion rule is a version of computation induction. Suppose E is an environment $\{p_i(\bar{r}_i, \bar{x}_i) \leftarrow S_i \mid i=1,...,n\}$. Also suppose for i=1,...,n and for some statement S that $\Delta_i[S]$ is a formula of the form

$$\left(\bigwedge_{j=1}^{m_i} \forall \bar{r}_i \, \forall \bar{v}_i \, (H_{i,j} \rightarrow \{U_{i,j}\} \; S \; \{V_{i,j}\}) \right),$$

where $\bar{r}_i$ is the list of procedure names in the formal list of p_i, and $\bar{v}_i$ is a list of environment variables. H and $H_{i,j}$ are formulas in which $p_1,...,p_n$ do not appear free. Under these assumptions, the Recursion Rule is as follows:

$$\frac{H \rightarrow \left(\left(\bigwedge_{i=1}^{n} \Delta_i[p_i(\bar{r}_i, \bar{x}_i)] \right) \rightarrow \left(\bigwedge_{i=1}^{n} \Delta_i[S_i] \right) \right)}{H \rightarrow \left(\bigwedge_{i=1}^{n} \Delta_i[E|p_i(\bar{r}_i, \bar{x}_i)] \right)}$$

4.2.3. Proofs

A formula H is said to be provable, written $\vdash H$, if it is an axiom or it can be derived from the axioms by applying rules of inference. More generally, we say that a formula H is provable from a set of assumptions Γ, written $\Gamma \vdash H$, if H can be derived using the added assumptions Γ. As was mentioned earlier, the axiom system is independent of the particular interpretation. To prove that a formula is true in a particular interpretation, one would prove it using the first-order theory of the interpretation as an assumption.

4.2.4. A Derived Axiom Scheme

In this section we introduce a useful set of formulas that can be derived from the axioms, and show that the formulas are derivable. We will refer to the fact that these formulas are provable in the example and the completeness proof. If C is a first-order formula whose only free variables are environment variables and H is any formula, then we define $C \rightsquigarrow H$ to be an abbreviation for a formula by induction on the structure of H:

1. $C \rightsquigarrow H \stackrel{def}{=} C \rightarrow H$, if H is a first-order formula.

2. $C \rightsquigarrow \{U\}\ S\ \{V\} \stackrel{def}{=} \{U \wedge C\}\ S\ \{V \wedge C\}$.

3. $C \rightsquigarrow (H1 \wedge H2) \stackrel{def}{=} (C \rightsquigarrow H1) \wedge (C \rightsquigarrow H2)$.

4. $C \rightsquigarrow (H1 \rightarrow H2) \stackrel{def}{=} (C \rightsquigarrow H1) \rightarrow (C \rightsquigarrow H2)$.

5. $C \rightsquigarrow (\forall v\ H) \stackrel{def}{=} \forall v'\ (C \rightsquigarrow H[v'/v])$,
   where v' is not free in H or C.

6. $C \rightsquigarrow (\forall q\ H) \stackrel{def}{=} \forall q\ (C \rightsquigarrow H)$.

It is straightforward to show by induction on the structure of H that $C \rightsquigarrow H$ is semantically equivalent to $C \rightarrow H$. We will use the formula $C \rightsquigarrow H$ to syntactically distribute the formula C to all the first-order parts of H. This will be useful in the course of the completeness proof. Since the formula $H \rightarrow (C \rightarrow H)$ is semantically true, it follows that the equivalent formula, $H \rightarrow (C \rightsquigarrow H)$, is true. We now show that this formula is provable in the axiom system.

Lemma. $\vdash H \rightarrow (C \rightsquigarrow H)$.

Proof.[6] We will first show that the following two formulas

(a) $\neg C \rightarrow (C \rightsquigarrow H)$

(b) $C \rightarrow (H \leftrightarrow (C \rightsquigarrow H))$

are provable. We will use induction on the structure of H, proving (a) and (b) simultaneously. All cases are completely straightforward and require only first-order reasoning, except when H is of the form $\{U\}\ S\ \{V\}$.

For part (a) in this case, first note that $\{False\}\ S\ \{False\}$ is provable for all S, by Ax 1 and 9.

Next, note that by Ax 0,

$\vdash \neg C \rightarrow \forall \bar{x}((U \wedge C) \supset False)$,

$\vdash \forall \bar{x}(False \supset (V \wedge C))$,

where $\bar{x}$ is the list of ordinary variables free in U, V.

---

[6] The proof of this lemma may be omitted on the first reading.

These formulas are provable by Ax 0 because they are valid wffs.
Here, we have explicitly quantified over $\bar{x}$ in order to form
formulas that match the hypotheses of Ax 7.

From Ax 7 and first-order reasoning, we have

$$\vdash (\neg C \rightarrow \forall\bar{x}((U \wedge C) \supset \text{False}) \wedge \forall\bar{x}(\text{False} \supset (V \wedge C)) \wedge \{\text{False}\}\ S\ \{\text{False}\}) \rightarrow (\neg C \rightarrow \{U \wedge C\}\ S\ \{V \wedge C\}).$$

Thus we can conclude $\vdash \neg C \rightarrow \{U \wedge C\}\ S\ \{V \wedge C\}$, as required.

For part (b), note that

$$\vdash C \rightarrow \forall\bar{x}(U \supset (U \wedge C)),$$

$$\vdash C \rightarrow \forall\bar{x}((V \wedge C) \supset V).$$

Thus, $\vdash C \rightarrow (\{U \wedge C\}\ S\ \{V \wedge C\} \rightarrow \{U\}\ S\ \{V\})$, by Ax 7.

From Ax 9, we get $\vdash \{U\}\ S\ \{V\} \rightarrow \{U \wedge C\}\ S\ \{V \wedge C\}$.

Thus $\vdash C \rightarrow (\{U\}\ S\ \{V\} \leftrightarrow \{U \wedge C\}\ S\ \{V \wedge C\})$, completing this case of (b).

Now we can show that $H \rightarrow (C \rightsquigarrow H)$ is provable.

From formula (a), we get $\vdash H \wedge \neg C \rightarrow (C \rightsquigarrow H)$.

From (b) we get $\vdash H \wedge C \rightarrow (C \rightsquigarrow H)$.

By propositional reasoning, it follows that $\vdash H \rightarrow (C \rightsquigarrow H)$. □

In the completeness proof, we will refer to the formula proved in this lemma as derived Axiom 13:

Ax 13. $H \rightarrow (C \rightsquigarrow H)$, provided C is a first-order formula whose only free variables are environment variables.

4.2.5. Soundness of the Axiom System

In this section we show that the axiom schemes and rules of inference presented in the previous section are sound; i.e. if $Th(I) \vdash H$ then $I \models H$, for any interpretation I and formula H. We will concentrate on proving the soundness of the recursion rule R4 here, because it is the main new element of the axiom system.

The axioms system can be considered in several parts. There are a number of axioms that do not depend on the details of procedure calls and that are familiar from other Hoare axiom systems. The axiom for assignment statement is one such axiom. The soundness of these axioms has been previously established in the literature with respect to operational definitions of the langauge that are the same as ours except perhaps for handling of procedures. There are axiom schemes such as Axiom 9, which apply to an arbitrary statement. The only property of procedure calls that is needed for these axioms to remain sound is that a procedure call $p(\bar{r}, \bar{x})$ in L4 can only change the values of the variables that appear as actual parameters in $\bar{x}$. It is clear that our semantics has this property. Finally, our axiom system permits first-order reasoning about formulas. It is clear that such reasoning is sound for our semantics of formulas.

We will now prove the soundness of the recursion rule R4. We must show that whenever the antecedent of R4 is true then the conclusion is also true. So suppose that E is an environment $\{p_i(\bar{r}_i, \bar{x}_i) \leftarrow B_i,\ i=1,...,n\}$, and suppose H and $H_{i,j}$ are formulas in which $p_1,...,p_n$ do not appear free. Then assume that the hypothesis of the recursion rule is valid in I, i.e.

$$(0)\ I \models H \rightarrow \Big(\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m_i} \forall \bar{r}_i, \bar{v}_i (\, H_{i,j} \rightarrow \{U_{i,j}\}\ p_i(\bar{r}_i, \bar{x}_i)\ \{V_{i,j}\}\,) \rightarrow \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m_i} \forall \bar{r}_i, \bar{v}_i (\, H_{i,j} \rightarrow \{U_{i,j}\}\ B_i\ \{V_{i,j}\}\,) \Big).$$

We want to show that for all environments F and environment valuations ξ that

$$(1)\ I,F,\xi \models H \rightarrow \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m_i} \forall \bar{r}_i, \bar{v}_i (\, H_{i,j} \rightarrow \{U_{i,j}\}\ E \mid p_i(\bar{r}_i, \bar{x}_i)\ \{V_{i,j}\}\,).$$

So suppose

$$(2)\ I,F,\xi \models H$$

(otherwise the result is immediate). Thus we must show

$$(3)\ I,F,\xi \models \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m_i} \forall \bar{r}_i, \bar{v}_i (\, H_{i,j} \rightarrow \{U_{i,j}\}\ E \mid p_i(\bar{r}_i, \bar{x}_i)\ \{V_{i,j}\}\,).$$

We can suppose without loss of generality that $p_1,...,p_n$ do not appear in F (otherwise we could just rename these variables, which are bound in E). Let $F_m = F \cup E_m$, where E_m is the environment defined in section 3.2, that gives the m^{th} approximation of the environment E. We will show by induction on m that for all m,

$$(4)\ I,F_m,\xi \models \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m_i} \forall \bar{r}_i, \bar{v}_i(\ H_{i,j} \rightarrow \{U_{i,j}\}\ p_{i,m}(\bar{r}_i, \bar{x}_i)\ \{V_{i,j}\}\ ).$$

By a straightforward argument, which is carried out below, we can show that no matter how the procedures $\bar{r}_i$ are declared in F, we have

$$(5)\ \mathcal{M}_{I,F}(E|p_i(\bar{r}_i, \bar{x}_i)) = \cup_m\ \mathcal{M}_{I,F_m}(p_{i,m}(\bar{r}_i, \bar{x}_i)).$$

From this, it can easily be seen that the truth of (4) for all m implies that (3) is true. For suppose that (3) is false. Then there must be some choice of i, j, $\bar{r}_i$, and $\bar{v}_i$ such that in I and F, $H_{i,j}$ is true and $\{U_{i,j}\}\ p_i(\bar{r}_i, \bar{x}_i)\ \{V_{i,j}\}$ is false. Thus there must be some $(\sigma,\sigma') \in \mathcal{M}_{I,F}(E|p_i(\bar{r}_i, \bar{x}_i))$ for which the pca is false. But by (5), there must be some value of m such that $(\sigma,\sigma') \in \mathcal{M}_{I,F_m}(p_{i,m}(\bar{r}_i, \bar{x}_i))$, and hence (4) must be false for this value of m. This shows that (4) is sufficient to prove (3).

Proving (4) for $m = 0$ is trivial, since in F_0 we have $p_i(\bar{r}_i, \bar{x}_i) \leftarrow$ error. Assume (4) holds for $m = N-1$. We now show it holds for $m = N$. It clearly suffices to show for all choices of F and ξ that

$$(6)\ I,F_N,\xi \models \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m_i} \forall \bar{r}_i, \bar{v}_i(\ H_{i,j} \rightarrow \{U_{i,j}\}\ p_{i,N}(\bar{r}_i, \bar{x}_i)\ \{V_{i,j}\}\ ).$$

Without loss of generality, we can assume

$$(7)\ I,F_N,\xi \models \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m_i} H_{i,j}.$$

Under this assumption, we must show

$$(8)\ I,F_N,\xi \models \{U_{i,j}\}\ p_{i,N}(\bar{r}_i, \bar{x}_i)\ \{V_{i,j}\}.$$

Now, we use the inductive hypothesis (4) for $m = N-1$. Previously, we have assumed that the hypothesis of the recursion rule is valid (0), and that H is true in F,ξ (2). Since $p_1,...,p_n$ are not free in H, the formula to the right of H in line (0) is true for all $p_1,...,p_n$. Since (4) is true for $m = N-1$, we get

$$(9)\ I,F_{N-1},\xi \models \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m_i} \forall \bar{r}_i, \bar{v}_i(\ H_{i,j} \rightarrow \{U_{i,j}\}\ B_i[p_{1,N-1}/p_1, ..., p_{n,N-1}/p_n]\ \{V_{i,j}\}\ ).$$

From (7) and the fact that $p_1,...,p_n$ are not free in $H_{i,j}$, we get

(10) $I,F_{N-1},\xi \models \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m_i} H_{i,j}$.

Using (9) and (10), we can conclude

(11) $I,F_{N-1},\xi \models \{U_{i,j}\}\ B_i[p_{1,N-1}/p_1\ ,\ ...,\ p_{n,N-1}/p_n]\ \{V_{i,j}\}$.

We will show

(12) $\mathcal{M}_{I,F_{N-1}}(B_i[p_{1,N-1}/p_1\ ,\ ...,\ p_{n,N-1}/p_n]) = \mathcal{M}_{I,F_N}(p_{i,N}(\bar{r}_i, \bar{x}_i))$.

Line (8) follows immediately from (11) and (12), so the proof of (12) will complete the inductive step of our proof.

To complete the details of this proof, we first prove the equivalence (5), and then prove (12).

The proof that (5) $\mathcal{M}_{I,F}(E|p_i(\bar{r}_i, \bar{x}_i)) = \cup_m \mathcal{M}_{I,F\cup E_m}(p_{i,m}(\bar{r}_i, \bar{x}_i))$ follows from two easy lemmas.

First, for any two statements A1 and A2, we will write A1 $\leq$ A2 if $\mathcal{M}_{I,E}(A1) \subseteq \mathcal{M}_{I,E}(A2)$ for all interpretations I and procedure environments E. Let us say that $\widetilde{F\cup E_n}|\widetilde{A}$ is a <u>variant</u> of $F\cup E|A$ if all occurrences of p_i in F and A have been replaced by $p_{i,k}$, for some $k \leq n$. Note that we do not necessarily use the same value of k in $p_{i,k}$ to replace different occurrences of p_i in $F\cup E|A$. We say that the <u>level</u> of $\widetilde{F\cup E_n}|\widetilde{A}$ is k if k is the least subscript s of a $p_{i,s}$ in either $\widetilde{F}$ or $\widetilde{A}$.

The following lemma is a consequence of the fact that E_n defines $p_{i,k}$ to be a procedure that is less defined than p_i is in environment E.

Lemma 1. $\forall k\ \forall A\ \forall F\ \forall n\ \text{Approx}^k_\emptyset(F\cup E|A) \geq \text{Approx}^k_\emptyset(\widetilde{F\cup E_n}|\widetilde{A})$ if $F\cup E|A$ is a progam and $\widetilde{F\cup E_n}|\widetilde{A}$ is a variant of $F\cup E|A$.

Proof. By induction on k and a subinduction on the structure of A. All cases are easy. □

We now give a second lemma which says that if a variant has level k, so that all second subscripts s in procedure names $p_{i,s}$ in the variant are at least k, then the k^{th} approximation of the variant is at least as defined as the k^{th} approximation of the original statement. The intuitive reason for this is that each of the procedures $p_{i,s}$ is at

least the k^{th} approximation of p_i.

Lemma 2. $\forall k\ \forall A\ \forall F\ \forall n$ if level($\widetilde{F \cup E_n}|\widetilde{A}$) $\geq$ k and $\widetilde{F \cup E_n}|\widetilde{A}$ is a variant of $F \cup E|A$ and $F \cup E|A$ is a program, then $Approx^k_{\emptyset}(\widetilde{F \cup E_n}|\widetilde{A}) \geq Approx^k_{\emptyset}(F \cup E|A)$.

Proof. Another easy induction on structure. □

Note that if S is a statement with no free procedures, then $\mathcal{M}_{I,E}(S)$ is independent of E. In the remainder of the proof, we will write $\mathcal{M}_I(S)$ instead of $\mathcal{M}_{I,E}(S)$ to denote that the meaning does not depend on an environment E.

Now, $\mathcal{M}_{I,F}(E|p_i(\bar{r}_i, \bar{x}_i))$

$= \cup_k \mathcal{M}_I(Approx^k_{\emptyset}(F \cup E|p_i(\bar{r}_i, \bar{x}_i)))$

$\leq \cup_k \mathcal{M}_I(Approx^k_{\emptyset}(F \cup E_k|p_{i,k}(\bar{r}_i, \bar{x}_i)))$ (by Lemma 2, since this has level k)

$\leq \cup_k \mathcal{M}_{I,F \cup E_k}(p_{i,k}(\bar{r}_i, \bar{x}_i))$.

But, $\mathcal{M}_{I,F \cup E_k}(p_{i,k}(\bar{r}_i, \bar{x}_i))$

$= \cup_m(\mathcal{M}_I(Approx^m_{\emptyset}(F \cup E_k|p_{i,k}(\bar{r}_i, \bar{x}_i)))$

$\leq \cup_m \mathcal{M}_I(Approx^m_{\emptyset}(F \cup E|p_i(\bar{r}_i, \bar{x}_i)))$ (by Lemma 1)

$= \mathcal{M}_{I,F}(E|p_i(\bar{r}_i, \bar{x}_i))$.

Therefore, $\mathcal{M}_{I,F}(E|p_i(\bar{r}_i, \bar{x}_i)) = \cup_k \mathcal{M}_{I,F \cup E_k}(p_{i,k}(\bar{r}_i, \bar{x}_i))$,
thus completing the proof of (5).

We now prove line (12), to complete the soundness proof.

$\mathcal{M}_{I,F \cup E_N}(p_{i,N}(\bar{r}_i, \bar{x}_i)) = \cup_k \mathcal{M}_I(Approx^k_{\emptyset}(F \cup E_N|p_{i,N}(\bar{r}_i, \bar{x}_i)))$

$= \cup_k \mathcal{M}_I(Approx^{k-1}_{\emptyset}(F \cup E_N|B_i[p_{1,N-1}/p_1, \ldots, p_{n,N-1}/p_n]))$

$= \cup_k \mathcal{M}_I(Approx^{k-1}_{\emptyset}(F \cup E_{N-1}|B_i[p_{1,N-1}/p_1, \ldots, p_{n,N-1}/p_n]))$
(since $p_{1,N}, \ldots, p_{n,N}$ are not free in either F or B_i)

$= \mathcal{M}_{I,F \cup E_{N-1}}(B_i[p_{1,N-1}/p_1, \ldots, p_{n,N-1}/p_n])$ □

4.3. Aliasing in Procedure Calls

We will say that a procedure call has aliasing if there is a ground variable that appears more than once in the call. It is well known that aliasing is a potential source of difficulty in formal reasoning. For example, Axiom 11 would be unsound without the requirement that the variable mapping θ be injective. The restriction gives a sound axiom system but leaves incompleteness in the special case of programs that have aliasing. We describe below a simple method for removing aliasing by transforming any program into an equivalent one with a similar structure, but without aliasing. Our reasons for taking this approach are that aliasing introduces additional complexity in reasoning, but in practice it is an unusual and exceptional case. The difficulties introduced by aliasing are separate from the main focus of this paper, which is reasoning about programs with higher type procedures.

Aliasing has been dealt with previously, for example in [Ol81/84]. There, the approach taken is to introduce axioms for reasoning separately about each case of aliasing of a procedure. Calls on a given procedure q may be divided into equivalence classes based on the partitioning of the ground parameters. For instance, if q has two parameter positions for ground variables, then there are two equivalence classes of calls: calls with two different actual ground parameters, and calls with the same actual appearing twice. Thus procedure calls can be divided into equivalence classes such that all calls in an equivalence class are identical up to injective renaming of variables. Intuitively, the method of reasoning about aliasing in [Ol81/84] is to prove one assertion for each equivalence class of calls, and then to use injective renaming to reason about other calls in the equivalence class. We feel that such a method could be incorporated in our axiom system in a straightforward way. However, for the purposes of our presentation, it is more convenient to assume that aliasing is analyzed beforehand, and to only work in the axiom system with programs having no aliasing.

In Appendix 1, we briefly sketch a method for removing aliasing. The basic idea is to replace each procedure declaration with a set of new declarations, where there is one new declaration corresponding to each case of aliasing. All procedure calls are modified to use one of the new procedures. The resulting procedure calls have no aliasing.

In view of this result, and our feeling that aliasing is an unusual situation, we will assume in the remainder of the paper that programs are presented in a form that is free from aliasing.

5. An Example

In this section, we demonstrate the use of the axiom system with a simple example. We feel that examples such as the one presented here and others that we have studied, show that the axiom system gives a way of reasoning that is very natural for actual use.

The reader may find it helpful to study the proof of the example carefully before proceeding to the completeness proof. In particular, we discuss the problem of how to specialize general assertions about procedures in order to reason about particular calls. In the completeness proof, the process of specializing general assertions involves some technical steps that are discussed here in a simpler form.

Let π be the following program:

```
Begin
    inc(w)  ←  w := w+1;
    p(r, x)  ←
        Begin
            twice(y)  ←  Begin r(y); r(y) End;
            If x = 0 Then r(x) Else Begin x := x − 1; p(twice, x) End;
        End;
    p(inc, z)
End.
```

The program is interpreted over the natural numbers with the standard arithmetic operations (for subtraction, note $0 - 1 = 0$).

We will show that $\vdash \{z = z_0\}\ \pi\ \{z = 2^{z_0}\}$. Intuitively, if z_0 is the initial value of z, then π sets z to 2^{z_0} (providing $z_0 \geq 0$), by calling the procedure p recursively z_0+1 times. The procedure p decrements z by 1 and calls itself recursively on each of the first z_0 calls; on the z_0+1^{st} invokation of p, the formal procedure r is called.

On the first call of p, the actual procedure parameter is inc, a procedure that increments its argument by 1. When p calls itself recursively, the actual procedure parameter is twice, a procedure that has r free in its body; the procedure twice(y) calls r(y) two times. Thus if r(y) is a procedure that increments y by some constant value v, twice(y) will increment y by 2·v. It follows that on the i^{th} recursive call of p, the procedure parameter r(y) increments y by 2^{i-1}. On the z_0+1^{st} call of p, the procedure r(x) is called, and has the effect of setting z to 2^{z_0}.

Now let us see how to formalize this argument in the axiom system. We begin by defining assertions for each of the procedures in π. We define P, R, and Twice to be formulas for the procedures p, r, and twice, respectively.

$$R \stackrel{def}{=} \{w = w_0\}\ r(w)\ \{w = w_0+v\}$$

$$\text{Twice} \stackrel{def}{=} \{y = y_0\}\ \text{twice}(y)\ \{y = y_0+2\cdot v\}$$

$$P \stackrel{def}{=} \forall r\, \forall v\, (\{w = w_0\}\ r(w)\ \{w = w_0+v\} \rightarrow \{x = x_0\}\ p(r, x)\ \{x = v\cdot 2^{x_0}\})$$

Throughout the example, the variables v and v′ will be environment variables; all other ground variables will be ordinary. Intuitively, the formula R, which has free occurrences of r and v, says that r(w) increments w by some constant amount v. The environment variable v is used to define a relationship between the procedures. For instance, Twice asserts that the procedure twice(y), which calls r(y) two times, increments its argument by 2·v. The formula P uses universal quantification over r and v to make an assertion about the call p(r, x) for an infinite range of procedures. Intuitively, P asserts that if r(w) is a procedure that increments w by v (so that r satisfies the formula R on the left of the arrow), then p(r, x) sets x to $v\cdot 2^{x_0}$. In this way, we use a relationship between r and v to express the effect of the call p(r, x) for different procedures.

The main idea of the proof is to use the recursion rule to show that the formula P holds for the declaration of the procedure p. After this we instantiate the universal variables in P with the substitution [inc/r, 1/v]. On the left side of this instantiated formula is the pca $\{w = w_0\}\ \text{inc}(w)\ \{w = w_0+1\}$. Since the declaration of inc satisfies this pca, we can discharge it and deduce that $\{x = x_0\}\ p(\text{inc}, x)\ \{x = 2^{x_0}\}$ holds in the environment with p and inc. Then we simply rename x and x_0 to z and z_0 to complete the proof.

As just described, there is a simple way to specialize a general assertion about a procedure in order to reason about a particular call. The formula P specifies the behavior of p(r, x), where r is any procedure that increments its argument by a constant. In order to specialize the assertion P to work for a call p(q, x), where q(w) increments its argument by a constant, say c, we simply instantiate P with the substitution [q/r, c/v] and then discharge the pca on the left side of the arrow. In the completeness proof, we must also specialize general assertions to particular calls, but we must use a different method. The problem is that, in general, there is no term that can play the role of the constant c. In the general case, we cannot find a term to substitute for v, but we can find a first-order formula that defines the possible values of v. So, we use a slightly different sequence of steps. In order to explain this aspect of the

completeness proof, we have shown how part of the example would be done in both ways.

We now define abbreviations for the procedure environments that we will need to refer to in the proof. If procs is a subset of {inc, p, twice}, let E_{procs} be the environment containing the named procedures of π.

The main part of the proof is reasoning about the body of p in order to apply the recursion rule. In order to apply the recursion rule, we will show

$$\vdash P \wedge R \rightarrow \{x = x_0\}\ E_{\{twice\}} \mid \text{If } x = 0 \text{ Then r(x) Else Begin } x := x - 1;\ p(twice, x) \text{ End}\ \{x = v \cdot 2^{x_0}\}.$$

Observe that if initially $x = x_0 = 0$, then the body of p simply calls r(x). This case of the If statement can be proven directly from the assumption R by standard methods, because the postcondition in R is $x = x_0 + v$, and in this case, $x_0 = 0$. Thus it is straightforward to show

$$\vdash R \rightarrow \{x = x_0 \wedge x_0 = 0\}\ r(x)\ \{x = v \cdot 2^{x_0}\}.$$

The other branch of the If statement involves the call p(twice, x); to reason about this call, we have to specialize the general formula P.

In order to reason about the call p(twice, x), we instantiate the universally quantified variables in P with the substitution [twice/r, 2·v/v] to get

$$\vdash P \rightarrow (\{w = w_0\}\ twice(w)\ \{w = w_0 + 2 \cdot v\} \rightarrow \{x = x_0\}\ p(twice, x)\ \{x = 2 \cdot v \cdot 2^{x_0}\}).$$

Applying Ax 11 to the formula Twice, and using first-order reasoning, we have

$$(1) \vdash P \wedge Twice \rightarrow \{x = x_0\}\ p(twice, x)\ \{x = v \cdot 2^{x_0 + 1}\}.$$

It is now straightforward to show

$$\vdash P \wedge R \wedge Twice \rightarrow \{x = x_0\}\ \text{If } x = 0 \text{ Then r(x) Else Begin } x := x - 1;\ p(twice, x) \text{ End}\ \{x = v \cdot 2^{x_0}\}.$$

Using rule R2 with the environment $E_{\{twice\}}$, we get

(2) $\vdash$ P $\wedge$ R $\wedge$ $E_{\{twice\}}|$Twice $\rightarrow$

$\{x = x_0\}$
$E_{\{twice\}}$ | If x = 0 Then r(x) Else Begin x := x − 1; p(twice, x) End
$\{x = 2^{x_0+1}\}$.

The next step is to discharge the assumption $E_{\{twice\}}|$Twice. We show that if r satisfies R, then twice satisfies Twice, i.e. R $\rightarrow$ $E_{\{twice\}}|$Twice.

It is straightforward to show

$\vdash$ R $\rightarrow$ $\{y = y_0\}$ Begin r(y); r(y) End $\{y = y_0+2\cdot v\}$.

Thus by first-order reasoning,

$\vdash$ R $\rightarrow$ (Twice $\rightarrow$ $\{y = y_0\}$ Begin r(y); r(y) End $\{y = y_0+2\cdot v\}$).

Applying R4, we can infer

$\vdash$ R $\rightarrow$ $E_{\{twice\}}|$Twice,

which allows us to discharge the assumption Twice.

Now, using first-order reasoning and rearranging the formula (2), we have

$\vdash$ P $\rightarrow$ (R $\rightarrow$
$\{x = x_0\}$
$E_{\{twice\}}$ | If x = 0 Then r(x) Else Begin x := x − 1; p(twice, x) End
$\{x = 2^{x_0+1}\}$).

By rule R1, we can universally quantify over r and v in this formula. Since neither r nor v is free in P, we can move the quantifiers in to get

$\vdash$ P $\rightarrow$ $\forall r\ \forall v$ (R $\rightarrow$
$\{x = x_0\}$
$E_{\{twice\}}$ | If x = 0 Then r(x) Else Begin x := x − 1; p(twice, x) End
$\{x = 2^{x_0+1}\}$).

This is in the form of the hypothesis of the recursion rule. Applying the recursion rule, we deduce

(3) $\vdash$ $\forall r\ \forall v$ ($\{w = w_0\}$ r(w) $\{w = w_0+v\}$ $\rightarrow$ $\{x = x_0\}$ $E_{\{p\}}|$p(r, x) $\{x = v\cdot 2^{x_0}\}$).

This is the general result we need for the procedure p.

For the procedure inc, it is straightforward to show

$\vdash$ $\{w = w_0\}$ $E_{\{inc\}}|$inc(w) $\{w = w_0+1\}$.

Finally, we specialize the formula of line (3) for the call p(inc, z). This proceeds as before, and we can show

$\vdash \{z = z_0\}\ E_{\{inc,\ p\}}|p(inc,\ z)\ \{z = 2^{z_0}\},$

as required to complete the example.

5.1. Specializing General Assertions

As we discussed at the beginning of the example, the completeness proof uses a different sequence of steps to specialize general assertions for reasoning about particular calls. In order to illustrate these steps, we will now re-derive line (1) from the example. Recall that line (1) is the result of specializing the general formula P in order to reason about the call p(twice, x).

The first step is to instantiate the universally quantified variables in P with the subsitution [twice/r, v'/v], where v' is a new environment variable, to get

$$\vdash P \rightarrow (\{w = w_0\}\ \text{twice}(w)\ \{w = w_0{+}v'\} \rightarrow \{x = x_0\}\ p(\text{twice}, x)\ \{x = v'{\cdot}2^{x_0}\}).$$

Using Axiom 13 (see 4.2.4) with the first-order formula $v' = 2{\cdot}v$ for C gives

$$(4) \vdash P \rightarrow (\{w = w_0 \wedge v' = 2{\cdot}v\}\ \text{twice}(w)\ \{w = w_0{+}v' \wedge v' = 2{\cdot}v\} \rightarrow \{x = x_0 \wedge v' = 2{\cdot}v\}\ p(\text{twice}, x)\ \{x = v'{\cdot}2^{x_0} \wedge v' = 2{\cdot}v\}).$$

Our next step is to derive the pca involving twice(w) from the formula Twice,

$$\vdash \text{Twice} \rightarrow \{w = w_0 \wedge v' = 2{\cdot}v\}\ \text{twice}(w)\ \{w = w_0{+}2{\cdot}v \wedge v' = 2{\cdot}v\},$$

by Ax 11 and Ax 9. Next, we use Ax7 to get

$$\vdash \text{Twice} \rightarrow \{w = w_0 \wedge v' = 2{\cdot}v\}\ \text{twice}(w)\ \{w = w_0{+}v' \wedge v' = 2{\cdot}v\}.$$

By first-order reasoning, we can use the assumption Twice in place of the pca for twice(w) in line (4),

$$\vdash P \wedge \text{Twice} \rightarrow \{x = x_0 \wedge v' = 2{\cdot}v\}\ p(\text{twice}, x)\ \{x = v'{\cdot}2^{x_0} \wedge v' = 2{\cdot}v\}.$$

Now, we use rule R3 to existentially quantify over v' on both sides of the rightmost pca,

$$\vdash P \wedge \text{Twice} \rightarrow \{\exists v'\,(x = x_0 \wedge v' = 2{\cdot}v)\}\ p(\text{twice}, x)\ \{\exists v'\,(x = v'{\cdot}2^{x_0} \wedge v' = 2{\cdot}v)\}.$$

Using Ax 7 to simplify the first-order formulas, we get

$\vdash P \wedge \text{Twice} \rightarrow \{x = x_0\}\ p(\text{twice}, x)\ \{x = v{\cdot}2^{x_0+1}\}$,

which is the formula from line (1) of the example.

6. Part 1: An Analysis of the Semantics of L4

Our goal in this section is to show that corresponding to any L4 program π, there exists an L4 program π^* without procedure parameters, that approximates π in an appropriate sense. In order to do this, we must first carefully analyze the semantics of L4.

6.1. Notation

This section defines some notation for sequences and substitutions that will be used in the remainder of the paper.

If $\tau = (\tau_1,...,\tau_n,\text{var},...,\text{var})$ is a procedure type, then $|\tau|$ is the number of elements of the sequence that are procedure types and $||\tau||$ is the number of elements of the sequence that are ground types.

Sequence notation. For any kind of identifier, superscripts are used to denote finite sequences of identifiers of that kind. For instance, for declared procedure identifiers, if θ is an ordered sequence of positive integers, $\theta_1,...,\theta_n$, then $\bar{p}^{\theta}$ stands for the sequence of declared procedure identifiers $p_{\theta_1},...,p_{\theta_n}$. As a special case $\bar{p}^n$, where n is a positive integer, stands for the sequence $p_1,...,p_n$, and $\bar{p}^0$ stands for an empty sequence. Sequences of other kinds of identifiers are defined similarly.

For ease of notation, after a superscripted sequence identifier has been introduced in a context, we may use just the identifier to refer to the sequence, provided it is clear from the context what the identifier refers to. For example, if $\bar{x}^n$ is used to stand for a sequence of n program variables, we may later write $\bar{x}$ to stand for the same sequence when the reference is clear.

When we have introduced a sequence such as $\bar{x}^n = x_1,...,x_n$, we will often want to introduce other related names. As mentioned above we will write $\bar{x}$ to stand for the sequence $\bar{x}^n$ where the reference is clear from the context. Sometimes we need another *single variable* that is similar to the elements of the sequence $\bar{x}$; we may write x_0 in this case to indicate a new variable that is not part of the sequence $x_1,...,x_n$.

If a symbol, say x, has been introduced, then we sometimes append a digit or a prime, as in x0, x′, to denote a new symbol related to x. For instance, if $\bar{x}$ is a sequence, then $\bar{x}0$ or $\bar{x}'$ may be used to indicate another sequence of the same type and length.

Substitutions. If x and y are variables, [y/x] is a substitution of y for x. Substitutions separated by "," are simultaneous. For instance, $[y_1/x_1, y_2/x_2]$ is a simultaneous substitution of y_1, y_2 for x_1, x_2. If $\bar{x}$ and $\bar{y}$ are sequences both of the same

length, say n, then $[\bar{y}/\bar{x}]$ is the simultaneous substitution $[y_1/x_1,...,y_n/x_n]$. Substitutions separated by ";" are sequential. For example, $P[y/x;\ z/y] \equiv P[z/x]$, provided P is a formula in which y does not appear free.

6.2. Semantic Equivalence of Declarations

Our analysis of the semantics of L4 programs will focus on properties of declarations having no free procedure names.

Definition. A closed declaration of L4, i.e. a declaration having neither free ground variables nor free procedure identifiers, will be called a closure.

We now introduce a notion of semantic equivalence of declarations that will play an important role in the first two lemmas. The following definition applies to closures.

Definition. If $\tau = (\bar{\tau}^{|\tau|}, \text{var}^{\|\tau\|})$ is a procedure type then two closed L4 declarations $q_0{:}d_0$ and $q_0{:}d_0'$ of type τ and having the same main procedure identifier q_0, are semantically equivalent in an interpretation I iff for all length $|\tau|$ sequences of closed L4 declarations, $q_1{:}d_1,...,q_{|\tau|}{:}d_{|\tau|}$, having types $\tau_1,...,\tau_{|\tau|}$ and having distinct main procedure identifiers,

$$M_I(\ \{d_0,d_1,...,d_{|\tau|}\}|q_0(\bar{q}^{|\tau|},\ \bar{x}^{\|\tau\|})\ ) \ =\ M_I(\ \{d_0',d_1,...,d_{|\tau|}\}|q_0(\bar{q},\ \bar{x})\ ).$$

Note that semantic equivalence of declarations is an equivalence relation.

We write $d \simeq_I d'$ to denote d is semantically equivalent to d' in I. Where the interpretation is clear from the context, we write $d \simeq d'$. □

We now introduce the syntactic operations on declarations that will be used in simulating the execution of programs. For any statement S and environment E, E|S denotes the statement Begin E; S End, which is the statement S in the environment E. We now define E|d, where d is a declaration.

If d = q(formal-list) ← body is a declaration and $E = \{d_1,...,d_n\}$ is an environment, then E|d is the declaration q(formal-list) ← E|body, where procedure names in formal-list are renamed to prevent them from clashing with any of the procedure names in $d_1,...,d_n$. We call the operation that takes d and E and produces E|d the environment binding operation. For convenience, we require that the renaming be done in a unique way, so that for any d and E, E|d is uniquely defined. □

There is one more operation on declarations, called the renaming operation. This operation is takes a declaration d and returns an equivalent declaration with a different main procedure identifier. It is used to simulate binding of procedure declarations to formal procedure identifiers.

Renaming operation. If d = q(formal-list) ← body is a closed declaration and r is a formal procedure identifier having the same type as q, then we define r ← d to be the declaration, r(formal-list) ← {d}|q(formal-list). With this declaration, r is a procedure that simply calls the procedure q. (Note that because q has a higher type than any of the formal parameters, neither q nor r can appear in the formal-list.) □

A procedure call statement in L4 may pass either declared or formal procedures as parameters. In order to simulate the case of passing a declared procedure, we form a closure from the declaration of the procedure and the declarations of all procedures in its environment. To simulate a procedure call that passes a formal procedure identifier as an actual parameter, the simulation passes the closure for the formal parameter. This closure must have been formed from declared procedures at an earlier step of execution of the program. Thus, at any point in the simulation, if there are formal procedure identifiers visible, the simulation will have a closure representing the value of each one.

Definition. Let Closures be the set of all closures over the fixed signature Σ. For each procedure type τ, let Closures(τ) be the set of closures of type τ. If T is a set of procedure types, then Closures(T) is $\cup_{\tau \in T}$Closures(τ).

Lemma 1. Let I be an interpretation of a finite signature Σ, and assume that I is bounded for L4. Then for each type τ, the relation $\simeq_I$ partitions Closures(τ) into a finite number of equivalence classes.

Proof. Recall that the declarations in Closures(τ) are closed, and no declaration has free ground variables. A procedure call in L4 can only change the state by changing the values of its ground parameters. Intuitively, the role of the higher type parameters in a procedure call in L4 is to select between different possible relations that the procedure call uses to modify the values of the ground parameters. A procedure type has a fixed number of ground parameters. So, the proof will proceed by showing that for any n, there are a finite number of distinct semantics of *programs* with the n free variables $\bar{x}^n$. Then it is straightforward to show by induction on the depth of procedure types that $\simeq$ partitions the declarations of each type in Closures into a finite number of equivalence classes.

For any finite signature Σ, and natural number n, there is a program of L4 that on input $\bar{a}^n \in dom(I)^n$, enumerates all of the elements of dom(I) that can be generated from $\bar{a}^n$ [GH83, Ur83]. (In fact, this construction requires only recursive procedures with ground parameters and no globals.) This implies that if an interpretation I is bounded for L4, then I is uniformly locally finite, meaning that for each natural number

n, there is a number b such that for all $\bar{a}^n \in dom(I)^n$, less than b elements of dom(I) can be generated from $\bar{a}^n$ using the constants and functions of I.

By a computable semantics on $dom(I)^n$, we mean a subset of $val_I \times val_I$ that can be the semantics of a program over I with the n free variables $\bar{x}^n$. We want to show that if I is bounded, then over the finite signature Σ, for each n, there are a finite number of computable semantics on $dom(I)^n$.

For $\bar{a}^n \in dom(I)^n$, let $I(\bar{a}^n)$ be the substructure of I generated by $\bar{a}^n$ and the constants and functions. Let $< \bar{c}^n \leftarrow \bar{a}^n, I(\bar{a}^n) >$ denote the structure $I(\bar{a}^n)$ expanded by adding n new constants, $c_1,...,c_n$, for the values $a_1,...,a_n$, respectively. On input $\bar{a}$, a program can be regarded as computing on the structure $< \bar{c} \leftarrow \bar{a}, I(\bar{a}) >$, because it can refer to the values of its input variables. Let us say for $\bar{a}^n, \bar{b}^n \in dom(I)^n$ that $< \bar{c} \leftarrow \bar{a}, I(\bar{a}) >$ and $< \bar{c} \leftarrow \bar{b}, I(\bar{b}) >$ are indistinguishable iff they are isomorphic without renaming the symbols of Σ or the new constant symbols $\bar{c}$.

Define an equivalence relation, $\bar{a}^n \equiv \bar{b}^n$, iff $< \bar{c} \leftarrow \bar{a}, I(\bar{a}) >$ is indistinguishable from $< \bar{c} \leftarrow \bar{b}, I(\bar{b}) >$. Since I is uniformly locally finite, there is a fixed set of variable-free Herbrand terms containing the constants and function symbols of I and $\bar{c}$, say, T, such that for all $\bar{a}^n \in dom(I)^n$, all of the values of $dom(< \bar{c} \leftarrow \bar{a}, I(\bar{a}) >)$ are given by the values of the terms T in $< \bar{c} \leftarrow \bar{a}, I(\bar{a}) >$. Since Σ has only a finite number of relation symbols, one can write a finite set of ground atomic formulas, such that the truth values of the formulas in a structure $< \bar{c} \leftarrow \bar{a}, I(\bar{a}) >$, completely determines it up to isomorphism. Hence the relation $\equiv$ divides $dom(I)^n$ into a finite number of equivalence classes.

Since a program of type Σ has exactly the same executions on all indistinguishable inputs, its semantics on inputs in an equivalence class of $\equiv$ can be represented by a finite set of terms. Let π be a program with the free variables $\bar{x}^n$, and $A \subseteq dom(I)^n$ be a set of indistinguishable inputs. Then there is a finite set of n-tuples of terms, $H_A = \{\bar{t}1^n(\bar{x}),...,\bar{t}k^n(\bar{x})\}$, for some k, such that

$$\forall\, \bar{a}^n \in A, \forall\, \bar{b}^n \in dom(I)^n,$$
$$(\sigma[\bar{x}^n \leftarrow \bar{a}], \sigma[\bar{x}^n \leftarrow \bar{b}]) \in M_I(\pi) \quad \text{iff}$$
$$\bar{b} = \bar{t}1(\bar{a}) \vee ... \vee \bar{b} = \bar{t}k(\bar{a}).$$

For example, if π is a deterministic program with free variables $\bar{x}^n$, then for each set $A \subseteq dom(I)^n$ of indistinguishable inputs, one of the following holds: Either π halts on inputs in A and there is a single n-tuple of terms $\bar{t}^n(\bar{x})$, such that the final values of the variables on input $\bar{a}^n$ is $\bar{t}(\bar{a})$, or π diverges on inputs in A.

Since I is bounded, there is a uniform depth bound d, such that for all A, all of the terms in H_A can be written with depth less than d. This means that there is a uniform bound on the number of n-tuples in H_A, for all equivalence classes. Therefore, there are a finite number of distinct semantics of programs with n free variables $\bar{x}^n$.

It is now simple to show by induction on the structure of types that the relation $\simeq$ partitions the set of closed L4 declarations of each type τ into a finite number of equivalence classes.

The base case is when τ contains only ground elements. Since there are only a finite number of semantics of programs with free ground variables $\bar{x}^{||\tau||}$, $\simeq$ divides the closed L4 declarations of type τ into a finite number of equivalence classes.

Induction step: Assume τ contains the procedure elements $\tau_1,...,\tau_{|\tau|}$, and $||\tau||$ ground elements. By hypothesis, $\simeq$ divides the closed L4 declarations of types $\tau_1,...,\tau_{|\tau|}$ into a finite number of equivalence classes. Suppose, however, that there were an infinite number of semantically distinct closed L4 declarations of type τ. By definition, $p_0{:}d_0$ and $p_0'{:}d_0'$ of type τ are distinct iff there exists a sequence of declarations $p_1{:}d_1,...,p_{|\tau|}{:}d_{|\tau|}$, with the correct types and distinct main identifiers, such that

$$\mathcal{M}_I(\ \{d_0,d_1,...,d_{|\tau|}\}|p_0(\bar{p}^{|\tau|},\ \bar{x}^{||\tau||})\ ) \neq \mathcal{M}_I(\ \{d_0',d_1,...,d_{|\tau|}\}|p_0'(\bar{p},\ \bar{x})\ ).$$

Observe that $\mathcal{M}_I(\ \{d_0,d_1,...,d_{|\tau|}\}|p_0(\bar{p},\ \bar{x}))$ is not changed if we replace $\bar{d}^{|\tau|}$ by a sequence of semantically equivalent declarations with the same types. Since there are only a finite number of equivalence classes of the relation $\simeq$ for each of these types, there can only be an infinite number of semantically distinct declarations of type τ if there is some choice of $p_1{:}d_1,...,p_{|\tau|}{:}d_{|\tau|}$, such that $\mathcal{M}_I(\ \{d_0,d_1,...,d_{|\tau|}\}|p_0(\bar{p},\ \bar{x})\ )$ takes on an infinite number of values, for different declarations d_0. But this is not possible, because in each case, these values are the semantics of programs having the same free variables $\bar{x}^{||\tau||}$. □

6.3. Encodings of Declarations

Much of the relative completeness proof is concerned with arithmetic encodings of declarations and of operations on declarations. We will be concerned with encoding sets of declarations of the form Closures(τ), for some fixed types τ. In a bounded interpretation, members of the infinite set Closures(τ) can be represented with a finite set of codes, because we only need to encode the equivalence class of closed declarations. This comment motivates the following definition:

Definition. If D is a set of closures and I is an interpretation, then a function ρ: D $\rightarrow$ **N** (**N** denotes the natural numbers) is called an encoding function for D in interpretation I iff ρ satisfies the following conditions:

1. If I is bounded, two declarations are mapped to the same number by ρ iff they are semantically equivalent in I,

$$\forall\, d_1, d_2 \in \text{closed(D)}\ \ (\rho(d_1) = \rho(d_2)\ \text{ iff }\ d_1 \simeq_I d_2).$$

2. If I is unbounded, ρ assigns each declaration in D a distinct number. Furthermore, ρ(d) is computable, in the usual sense, as a function of the sequence of characters in d. □

We now use encoding functions to define other semantic and syntactic relations on the code numbers of declarations. Intuitively, these relations will have the following semantics: ν_r is the renaming relation for the formal procedure name r, which when given a code for a closed declaration ρ(q(formals) $\leftarrow$ body), produces a code for a declaration with main procedure identifier r and having the same semantics as the original declaration. For each open declaration d with free procedures $q_1,...,q_n$ of types $\tau_1,...,\tau_n$, there is a binding relation β_d. When β_d is given n codes for closed declarations of $q_1,...,q_n$, it produces a code for the declaration $\{q_1,...,q_n\}|d$. The relation ι_τ is the interpreter relation that simulates procedure calls of type τ. Given a code for a closed declaration of type τ and codes for $|\tau|$ closed procedure parameters and $||\tau||$ initial values of ground parameters, it determines the possible final values of the ground parameters.

Lemma 2. Let I be an interpretation of Σ and ρ be an encoding function for Closures in I. Then for each formal procedure name r, 2.1 defines a unique partial function ν_r: **N** $\rightarrow$ **N**; for each declaration d with n free procedure names, 2.2 defines a unique partial function β_d: $\mathbf{N}^n \rightarrow \mathbf{N}$; and for each procedure type τ, 2.3 defines a unique relation $\iota_\tau \subseteq \mathbf{N}^{|\tau|+1} \times \text{dom(I)}^{||\tau||} \times \text{dom(I)}^{||\tau||}$.

Recall that $|\tau|$ is the number of procedure elements in the type τ, and $||\tau||$ is the number of ground elements in the type τ.

2.1. $\nu_r(i) = \rho(r(\text{formal-list}) \leftarrow \{d\}|q(\text{formal-list}))$, if there is a declaration d = q(formal-list) $\leftarrow$ body in Closures such that $\rho(d) = i$ and r is a formal procedure identifier having the same type as q; otherwise $\nu_r(i)$ is undefined.

2.2. $\beta_{d_0}(i_1,...,i_n) = \rho(\{d_1,...,d_n\}|d_0)$ if d_0 is a declaration with n free procedures of types $\tau_1,...,\tau_n$, and $i_k = \rho(d_k)$ for k = 1,...,n, where d_k is a closure of type τ_k; otherwise $\beta_{d_0}(i_1,...,i_n)$ is undefined.

2.3. If $q_0{:}d_0$ is a closure of type τ and $q_1{:}d_1,...,q_{|\tau|}{:}d_{|\tau|}$ is a sequence of closures such that the type of q_i is the same as the type of the i^{th} procedure type in τ, then for all $\bar{a}$, $\bar{b} \in \text{dom}(I)^{\|\tau\|}$,

$$\iota_\tau(\rho(d_0), \rho(d_1),...,\rho(d_{|\tau|}), \bar{a}, \bar{b})$$
$$\text{iff} \quad (\sigma[\bar{x} \leftarrow \bar{a}], \sigma[\bar{x} \leftarrow \bar{b}]) \in \mathcal{M}_I(\{d_0,d_1,...,d_{|\tau|}\} \mid q_0(\bar{q}^{|\tau|}, \bar{x}^{\|\tau\|})).$$

For values c_0 and $\bar{c}^{|\tau|}$ that are not encodings of declarations satisfying the conditions above, $\iota_\tau(c_0, \bar{c}, \bar{a}, \bar{b})$ is false for all $\bar{a}$, $\bar{b}$.

In essence, the pair of valuations $(\sigma[\bar{x} \leftarrow \bar{a}], \sigma[\bar{x} \leftarrow \bar{b}])$ is in the relational semantics of a call iff the interpreter relation ι_τ holds for the encodings of the procedures in the call and the values $\bar{a}$ and $\bar{b}$. The intuitive idea behind the interpreter relation as specified by 2.3 is that for each L4 program π, we will construct an L4 program called π^* without procedures passed as parameters, which simulates π by passing code values for closures. Where π has a call on a formal procedure, say r(q, x), the simulation π^* will have a statement of the form INTERP(r*, q*, x), where INTERP is a program that computes the interpreter relation ι using a representation of arithmetic in dom(I). The variables r* and q* are new ground variables that at any point in an execution of π^* will be set to the codes for closed declarations equivalent to the procedures r and q at the same point in a corresponding execution of π. Property 2.3 says that the semantics of a procedure call in the language involving certain procedure declarations holds for a pair of valuations iff the relation ι holds for codes of the declarations and the values of the free ground variables. The simulation is defined precisely later in the proof.

Proof. The lemma asserts that ν_r, β_d, and ι_τ are uniquely defined , given any encoding function function ρ for Closures in I. In the unbounded case, this is clear from examination of 2.1 to 2.3, because ρ assigns a unique number to each declaration.

In the bounded case, ρ maps closures in each equivalence class of the relation $\simeq_I$ into a distinct number. Consider property 2.2. We want to show that β_{d_0} is

consistently defined as a function by defining the values of $\beta_{d_0}(\rho(d_1),...,\rho(d_n))$. Suppose that the codes of two distinct closed declarations appear in the i^{th} argument position of β_{d_0}. If d_i and d_i' are two declarations with $\rho(d_i) = \rho(d_i')$, then d_i and d_i' are semantically equivalent in I. But then, for any choice of the other declarations $\{d_1,...,d_{i-1},d_{i+1},...,d_n\}$, the two declarations $\{d_1,...,d_i,...,d_n\}|d_0$ and $\{d_1,...,d_i',...,d_n\}|d_0$ must be semantically equivalent. Thus these two declarations must be assigned the same code by ρ. This shows that β_d, and similarly ν_r, are uniquely defined for each encoding function ρ.

The relation ι_τ is well defined for each encoding function ρ for the same reason. Whenever the codes of two semantically equivalent declarations appear in one of the argument positions of ι_τ, then the semantics of the statement on the right side of 2.3 is the same in both cases. □

From now on, ρ will be an encoding function and ν_r, β_d, and ι_τ will refer to relations satisfying the conditions in Lemma 2.

We now discuss how the relations of Lemma 2 can be computed by a restricted class of L4 programs having no procedures as parameters. We will proceed in two steps. First, we will consider a special class of programs having two kinds of ground variables and no procedures as parameters. The first kind of ground variable ranges over dom(I), and the operations on these variables are the functions and predicates of I. The second kind of variable ranges over enough of the natural numbers to be able to encode the closures needed for our simulation.

In the second step of the discussion, we will replace the natural numbers by elements of dom(I). If I is unbounded, this causes no difficulty, because we can simulate arithmetic in dom(I). But if I is bounded, only a finite number of values can be represented. Recall that our ultimate goal is to find a way to simulate a given L4 program with another program having no procedures as parameters. Since any given program that we would like to simulate can contain only a finite number of procedure types, it is sufficient to be able to encode Closures(T), where T is a finite set of procedure types. Recall that if I is bounded, an encoding function assigns a natural number to each equivalence class of closures. We have seen in Lemma 1 that for each procedure type there are only a finite number of equivalence classes. Therefore, for any finite set of procedure types T, Closures(T) can be encoded with a finite number of code values when I is bounded.

Let us define Codes(T), where T is a set of procedure types, to be the image of ρ on Closures(T). Without loss of generality, we can assume that Codes(T) is **N** when I is

unbounded, and [1 .. m], for some m, when I is bounded. In the unbounded case, program expressions of the code type are written over the similarity type {0, 1, +, −, ×, <, =}. In the bounded case, program expressions for the code type are written over a similarity type containing only the constants 1,...,m, and equality.

Henceforth, we will call programs with the operations just described programs over I with Codes.

Lemma 3. Let I be an interpretation of Σ, T be a finite set of procedure types, and ρ be an encoding function for Closures(T) in I. Then there are programs over I with Codes that compute the relations of Lemma 2 for the types in T, as described in 3.1 to 3.3 below. In the following, x, y are variables over Codes(T), and u, v are variables over dom(I).

3.1. For each closure d, there is a program Codes-BIND$_d(\bar{x}^n, y)$ that halts and sets y to $\beta_d(\bar{x}^n)$, if this value is defined; otherwise the program diverges.

3.2. For each formal procedure name r, there is a program Codes-RENAME$_r(x, y)$ that halts and sets y to $\nu_r(x)$ if this value is defined; otherwise the program diverges.

3.3. For each $\tau \in T$, there is a program Codes-INTERP$_\tau(x_0, \bar{x}^{|\tau|}, \bar{u}^{\|\tau\|}, \bar{v}^{\|\tau\|})$, that does the following possibly nondeterministic computation: it does not change the inputs x_0, $\bar{x}$, or $\bar{u}$, but it sets $\bar{v}$ to any possible value such that $\iota_\tau(x_0, \bar{x}, \bar{u}, \bar{v})$ holds, if there is such a value. More precisely:

For all valuations σ and values $\bar{a} \in \text{dom(I)}^{\|\tau\|}$, there is a computation of Codes-INTERP$_\tau(x_0, \bar{x}, \bar{u}, \bar{v})$ starting in valuation σ, that halts and sets $\bar{v}$ to $\bar{a}$ iff the relation $\iota_\tau(\sigma(x_0), \sigma(\bar{x}), \sigma(\bar{u}), \bar{a})$ holds.

Note: The following is a consequence of 3.3. If there is no value $\bar{a} \in \text{dom(I)}^{\|\tau\|}$ such that $\iota_\tau(\sigma(x_0), \sigma(\bar{x}), \sigma(\bar{u}), \bar{a})$ holds, then Codes-INTERP$_\tau(x_0, \bar{x}, \bar{u}, \bar{v})$ always diverges when started in valuation σ.

Proof.

3.1. In the unbounded case, Codes-BIND$_d$ is a simple arithmetic program on code numbers. In the bounded case, there is a program that looks up the values of $\bar{x}^n$ in a finite list of cases, and sets y to the corresponding value. We do not give a way to find this program effectively, but it is sufficient for our purposes to know that it exists.

3.2. The program for Codes-RENAME$_r$ is similar to Codes-BIND$_d$.

3.3. In the unbounded case, the program Codes-INTERP$_\tau$ is an interpreter that takes as inputs arithmetic values encoding the character strings for some declarations

and initial values of ground variables over dom(I), and simulates execution of the program in I. Given the set of declarations D, there is a constructive definition of Codes-INTERP$_\tau$ as an L4 program. However, the construction is somewhat complicated and there is a simpler way to see that the program Codes-INTERP$_\tau$ exists, using the concept of an *acceptable programming language* from [CGH83]. One of the properties of an acceptable programming language is that there is an effective algorithm, which, when given an effective encoding of a program in the language and a initial valuation of the ground variables, simulates the program one step at a time. For each program in an acceptable language, there is a finite set of variables that it may may examine and assign values to. Let us consider an acceptable language with ground variables w_1, w_2, ...

Consider simulation of a program π such that all of the variables examined or assigned by π are among $w_1,...,w_k$. The simulation begins with an initial valuation σ_0 that maps $\bar{w}^k$ to some set of values $\bar{a}^k$. At step i of the simulation, the algorithm constructs a finite set of k-tuples terms of type Σ. Suppose this set at step i is $\{\bar{t}1^k(\bar{x}),...,\bar{t}n^k(\bar{x})\}$, for some n. Then the possible values of w_j for $j = 1,...,k$, are given by $\{t1_j(\bar{a}^k),...,tn_j(\bar{a}^k)\}$, where $\bar{a}^k \in \text{dom(I)}^k$ is the vector of initial values of the variables. At each step of the simulation, the algorithm must evaluate a finite set of atomic formulas in the interpretation I. The free variables of these atomic formulas are evaluated in the initial valuation, σ_0. Then the algorithm determines effectively if the compuation can halt at that step and what the finite set of next steps is. For further details refer to [CGH83].

In order to carry out this simulation using an L4 program over I with Codes, the declarations and sets of terms will be encoded in arithmetic. For any finite signature, there is a recursive program over I with arithmetic, $H(\bar{x}, \bar{u}^k, v)$, that evaluates terms. When given an input $\bar{x}$ that is a code for a term $t(\bar{w}^k)$ and input $\bar{u}^k \in \text{dom(I)}^k$, H sets v to $t(\bar{u}^k)$. Using H, one can write a program that evaluates atomic formulas. Other details of the simulation are straightforward.

It is interesting to note that at no point in this simulation do we need to apply operations to values of dom(I) other than the operations in I. In particular, we do not need to assume the existence of a pairing operation on dom(I). For example, intuitively, the definition of an acceptable programming language implies that it is possible to simulate value stacks in a programming language by using coded sequences of terms.

In the bounded case, we have to see that there is a way to simulate procedure calls when declarations are encoded using the finite encoding. We observed in the proof of

Lemma 1 that in a bounded interpretation, for each n, there are a finite number of semantically distinct programs with n free variables. Using this observation we can construct the program Codes-INTERP$_\tau$ by induction on the depth of τ.

For the base case, let τ be a procedure type containing $||\tau||$ ground elements and no procedure elements. Assume that the semantically distinct declarations of type τ have codes 1,...,m, for some m. Let $\bar{d}^m$ be a sequence of declarations of type τ such that $\rho(d_i) = i$, for $i = 1,...,m$. Also let $\bar{q}^m$ be the sequence of main procedure identifiers in $\bar{d}$. Now we define π_i to be a program with free variables $\bar{v}^{||\tau||}$ that invokes the declaration d_i with actual parameters $\bar{v}^{||\tau||}$:

$$\pi_i = \{d_i\}|q_i(\bar{v}).$$

Finally, Codes-INTERP$_\tau(x_0, \bar{u}^{||\tau||}, \bar{v}^{||\tau||})$ is the program,

```
Begin
      v̄ := ū;
      if x_0 = 1 then π_1;
            . . .
      if x_0 = m then π_m
      else diverge;
End.
```

One can easily see that this program has the required properties.

Induction step. If τ contains procedure elements, the construction of Codes-INTERP$_\tau(x_0, \bar{x}^{|\tau|}, \bar{u}^{||\tau||}, \bar{v}^{||\tau||})$ is much like the base case except that the conditional statements also test the values of the $\bar{x}$, the codes for the procedure elements. As in the proof of Lemma 1, we use the fact that the semantics of calls on a procedure with a parameter of a procedure type τ' does not change when the actual parameter ranges over semantically equivalent closed procedures of type τ'.

As in the base case, Codes-INTERP$_\tau$ first copies the values of $\bar{u}$ to $\bar{v}$. Then it has a series of conditional statements. There are a finite number of combinations of the code parameters x_0 and $\bar{x}$. For each combination, Codes-INTERP$_\tau$ simply branches to a different program with free variables $\bar{v}$. Note that this construction is not effective, but it shows that for higher types a program Codes-INTERP$_\tau$ exists with the required properties. □

The next step is to take the programs from Lemma 3 and replace the variables and operations over Codes(T) with variables and operations over dom(I). Again, this is done by splitting into the bounded and unbounded cases. One small complication is

introduced when we change from arithmetic operations to operations on dom(I). We cannot assume that the interpretation I has constants and functions that can generate enough of the domain to simulate the required amount of arithmetic. Instead, we use an idea of simulating *bounded arithmetic* from [Li77, CGH83]. Each of the programs for simulating arithmetic in I will have a special input variable, b, in addition to the variables for the arithmetic operands and result. The program will use the initial value of b to try to generate enough of the domain to simulate the necessary amount of arithmetic. There will always be some initial value of b for which the simulation can work. If the program gets an initial value of b for which it cannot simulate enough arithmetic, it will diverge. Thus programs constructed by this simulation have the following property: for every halting computation of the program interpreted over I with arithmetic, there is a value of b such that the program interpreted purely over I halts (and whenever the programs purely over I halt, they produce the correct result). Whenever the program over I with arithmetic diverges, so does the program purely over I.

Lemma 4. Let I be an interpretation of Σ with $|I|>1$, let T be a finite set of procedure types, and ρ be an encoding function for Closures(T) in I. Then for some $k>0$, there is a set NUM $\subseteq$ dom(I)^k and a surjection ϕ: NUM $\rightarrow$ Codes(T) such that there are programs interpreted over I with properties 4.1 - 4.3. In the following, $\bar{b}^k$ is a sequence of k ground parameters. The programs do not change the values of any of the variables except for the output variable, y.

4.1. For each d in Closures(T) there is a (deterministic) program $\text{BIND}_d(\bar{b}, \bar{x}^{n\cdot k}, \bar{y}^k)$ such that

$$\exists \bar{u}_b \, (\, (\sigma[\bar{b} \leftarrow \bar{u}_b, \bar{x} \leftarrow \bar{u}_x, \bar{y} \leftarrow \bar{u}_y], \; \sigma[\bar{b} \leftarrow \bar{u}_b, \bar{x} \leftarrow \bar{u}_x, \bar{y} \leftarrow \bar{u}'_y]) \in \mathcal{M}_I(\, \text{BIND}_d(\bar{b}, \bar{x}, \bar{y}) \,) \,)$$

iff

$$\bar{u}_x \in \text{NUM}^n \wedge \bar{u}'_y \in \text{NUM} \wedge \phi(\bar{u}'_y) = \beta_d(\phi(u_{x,1}),\ldots,\phi(u_{x,n})).$$

4.2. For each formal procedure name r appearing in D, there is a (deterministic) program $\text{RENAME}_r(\bar{b}, \bar{x}^k, \bar{y}^k)$ such that

$$\exists \bar{u}_b \, (\, (\sigma[\bar{b} \leftarrow \bar{u}_b, \bar{x} \leftarrow \bar{u}_x, \bar{y} \leftarrow \bar{u}_y], \; \sigma[\bar{b} \leftarrow \bar{u}_b, \bar{x} \leftarrow \bar{u}_x, \bar{y} \leftarrow \bar{u}'_y]) \in \mathcal{M}_I(\, \text{RENAME}_r(\bar{b}, \bar{x}, \bar{y}) \,) \,)$$

iff

$$\bar{u}'_y \in \text{NUM} \wedge \bar{u}_x \in \text{NUM} \wedge \phi(\bar{u}'_y) = \nu_r(\phi(\bar{u}_x)).$$

4.3. For each $\tau \in$ T, there is a (deterministic for deterministic L4,

nondeterministic for L4 with Or statements) program $\text{INTERP}_\tau(\bar{b},\ \bar{w}^k,\ \bar{x}^{|\tau|\cdot k},\ \bar{y}^{\|\tau\|},\ \bar{z}^{\|\tau\|})$, such that the following condition holds

$$\exists \bar{u}_b\ (\ (\sigma[\bar{b} \leftarrow \bar{u}_b,\ \bar{w} \leftarrow \bar{u}_w,\ \bar{x} \leftarrow \bar{u}_x,\ \bar{y} \leftarrow \bar{u}_y,\ \bar{z} \leftarrow \bar{u}_z],$$
$$\sigma[\bar{b} \leftarrow \bar{u}_b,\ \bar{w} \leftarrow \bar{u}_w,\ \bar{x} \leftarrow \bar{u}_x,\ \bar{y} \leftarrow \bar{u}_y,\ \bar{z} \leftarrow \bar{u}'_z]\ )\ \in\ M_I(\text{INTERP}_\tau(\bar{b},\ \bar{w},\ \bar{x},\ \bar{y},\ \bar{z}))\ )$$
$$\text{iff}$$
$$\bar{u}_w \in \text{NUM}\ \wedge\ \bar{u}_x \in \text{NUM}^{|\tau|}\ \wedge\ \iota_\tau(\phi(\bar{u}_w),\ \phi(u_{x,1}),\ldots,\phi(u_{x,|\tau|}),\ \bar{u}_y,\ \bar{u}'_z).$$

Proof. We need to substitute programs that simulate arithmetic over I in place of the operations on natural numbers used in the programs of Lemma 3. In the bounded case, we will assume that the domain contains at least two elements, so that code numbers 0,...,m can be encoded by k-tuples of domain values, for some k. This encoding cannot depend on the program being able to refer to the k-tuples as constants or values generated from constants because we do not assume that the domain is Herbrand. An example of an encoding that can work is, let a k+2 tuple of domain values encode a k-bit binary number. The first two values of the tuple are the values for 0-bit and 1-bit. If these values are the same, or if the k+2 tuple contains values other that these two values, then say the k+2 tuple encodes the number 0. Otherwise, the k+2 tuple encodes the binary number given by the last k values. For any fixed k, it is simple to write programs that use this encoding of arithmetic.

In the unbounded case, we make use of the fact that for some k there is a program with k input variables that can set its variables to an unbounded number of valuations. Then, as in [Li77, CGH83], pairs of k-tuples can be used to encode integers using the unbounded program, where $(\bar{u}^k, \bar{v}^k)$ encodes the number n iff the program goes from the values $\bar{u}$ to $\bar{v}$ in n steps.

In both cases, the program will check whether certain inputs are in the set NUM, and will diverge for inputs not in the set. This is straightforward for the bounded case. In the unbounded case, there is a program in [CGH83] that halts if a k-tuple of domain elements represents a natural number and diverges otherwise. □

The programs of Lemma 4 are used in the proof in two ways. First, they show that if the interpretation I is expressive, then the interpreter relations are expressed by first-order formulas. This result plays an important role in Part 2. Also, we will use the programs to construct a simulation of procedures passed as parameters, using domain values to encode closures.

We need another simple expressibility result concerning the encoding function ρ and the representation of natural numbers by k-tuples of domain elements with ϕ and

NUM. For each procedure type τ, we show that there is a first-order formula $W_\tau(\bar{x}^k)$ that expresses membership in the set of domain values that represent codes of declarations in Closures(τ).

Lemma 5. Let I be an expressive interpretation, T be a finite set of procedure types, and ρ be an encoding function for Closures(T) in I. Let ϕ and NUM $\subseteq$ dom(I)^k satisfy the conditions of Lemma 4, and for each procedure type τ in T, let WELL$_\tau$ $\subseteq$ dom(I)^k be the set of all $\bar{x} \in$ NUM such that there is a declaration d in Closures(τ) for which $\rho(d) = \phi(\bar{x})$. Then there is a first-order formula $W_\tau(\bar{x}^k)$ that expresses membership in WELL$_\tau$, i.e.

$I \models W_\tau(\bar{x})$ iff $\bar{x} \in$ WELL$_\tau$.

Proof. Again, the proof splits into cases. In a bounded interpretation, Codes(τ) is a finite set. For each natural number n, there is a program that halts on input $\bar{x}^k$ iff $\phi(\bar{x}) = n$. Thus, in an expressive interpretation, we can express membership in the subset of dom(I)^k representing Codes(τ). In an unbounded interpretation, there is an infinite set of natural numbers in Codes(τ) for declarations of each type, but because the encoding function ρ must be a computable function of the text of a declaration, it is clear that for each type τ there is a program that on input $\bar{x}^k$ halts iff $\phi(\bar{x})$ is a code for a declaration Closures(τ). □

6.4. Simulating Procedures Passed as Parameters

We will present a set of program transformations that start with a program π and the programs from Lemma 4, and produce a new program called π^* that has no procedures as parameters, but is still in L4. Let $\bar{x}$ be the sequence of free variables in π. Then π^* has semantics in I that approximates that of π in the following sense: π^* has a new free variable b not free in π such that

$$(\sigma[\bar{x} \leftarrow \bar{u}], \sigma[\bar{x} \leftarrow \bar{u}']) \in \mathcal{M}_I(\pi)$$
$$\text{iff}$$
$$\exists z\ ((\sigma[b \leftarrow z, \bar{x} \leftarrow \bar{u}], \sigma[b \leftarrow z, \bar{x} \leftarrow \bar{u}']) \in \mathcal{M}_I(\pi^*)).$$

The program π^* never changes the value of b, it just uses it as an input generate a certain amount of arithmetic.

If σ is a valuation, and Q is a first order formula that does not have a free occurrence of b, then $I,\sigma \models SP[\pi;\ Q]$ iff $\exists z\ (I,\sigma[b \leftarrow z] \models SP[\pi^*;\ Q])$. Thus in an expressive interpretation,

$$I \models SP[\pi;\ Q] \equiv \exists b\ SP[\pi^*;\ Q].$$

The statements within π^* have additional semantic relations to π that we will discuss later.

In the remainder of the paper, we assume without loss of generality that programs satisfy the following conditions:

1. All declarations in π have distinct main procedure identifiers.

2. No formal procedure identifier appears more than once in any formal part in π. In other words, a formal procedure identifier cannot be bound in more than one place in π.

3. All names of formal ground variables are distinct from local variables.

Any program can be easily put in this form by renaming, if necessary. Clearly, $I \models \{U\}\ \pi\ \{V\} \leftrightarrow \{U\}\ \pi'\ \{V\}$, where π' is the result of renaming bound variables (declared procedure names, names in the formal lists of procedures, and local variable names) in π. This renaming can be formally applied in the axiom system by using Ax 12 once at the beginning of a proof.

We now define π^* by induction on the structure of programs. The first five cases are simple:

1. (x := e)* = x := e,

2. (If b Then S1 Else S2)* = If b Then S1* Else S2*,

3. (S1 Or S2)* = S1* Or S2*,

4. (S1; S2)* = S1*; S2*,

5. (Begin var x; S End)* = Begin var x; S* End.

The definitions of S* when S is a procedure call or a block with a procedure declaration do most of the work of the simulations and are more complicated. Unlike the first five cases above, the translations of these statements are given relative to the entire program. In order to understand the transformation of procedure calls and declarations, it will be helpful to keep in mind that the transformation accomplishes two different things. First, it replaces procedures passed as parameters by ordinary ground variables passed as parameters. This is the most complicated aspect of the transformation, because it requires the addition of programs to dynamically compute the code values and other programs to interpret the values. The other aspect of the transformation is the removal of references to globals. In L4, a procedure declaration can have free occurrences of formal procedure names, but of course, no free occurrences of ground variables. The completeness proof will use the fact that the transformed program has no free occurrences of ground variables in its procedure declarations. In order to translate global references to formal procedures, we will expand the list of ground parameters in declarations in the transformed program. These declarations will have new ground parameters to pass along the codes for the formal procedures that were global in the original program.

We will need the familiar notion of the scope of a procedure name: If p is declared in the environment E in Begin E; S End, then the scope of p is the entire statement Begin E; S End. If r_i is formal procedure name in the list $\bar{r}$ in a declaration $p(\bar{r}, \bar{x}) \leftarrow$ stmt, then the scope of r_i is the statement stmt.

We say that a declared procedure name p is in the scope of another declared procedure name p′ in π iff the declaration of p is in the scope of p′; p is in the scope of a formal procedure name r in π iff the declaration of p is contained within the declaration that has r in its formal list. Intuitively, the formal procedure names that have meanings in the body of procedure p are the ones in the formal list of p and the ones that have p in their scope. We will translate a procedure p into a new procedure p*, with code parameters for these two sets of formal procedures.

Let $r_1,...,r_\lambda$ be the formal procedure identifiers appearing in π, where λ is the number of distinct formal procedure identifiers in π. For each of the formal procedure parameters r_i, $i=1,...,\lambda$, the translation introduces a new ground variable, r_i^*. The purpose of r_i^* is to hold a ground value that is a code of a closure corresponding to the procedure r_i. Roughly speaking, the idea is that r_i^* will be set dynamically during the computation of π^* to a value that represents the meaning of r_i at the corresponding step in a computation of π.

In order to represent the natural numbers for the codes as domain values in I, it is necessary to use a sequence of domain values of a fixed length that depends on I. Thus the variables r_i^* are actually sequences of k ground variables, for some k. Since we never need to refer to the individual variables in such a sequence, we will refer to the entire sequence by a single variable name, such as r_i^*. Assignment statements that assign to code variables, and code variables used as formal and actual parameters in programs, have the obvious meanings.

Notation. We now define certain sequences of the r and r* variables that will be used for the rest of the proof. Intuitively, r_i^* is a code variable for the formal procedure r_i. If p is a declared procedure name in π, then $\bar{r}^{f\,p}$ is the list of formal procedure names in the formal list of p, and $\bar{r}^{*f\,p}$ is the list of r_i^* such that r_i is in the formal list of p. We define $\bar{r}^{g\,p}$ to be the list of formal procedure names that have p in their scope, and similarly, $\bar{r}^{*g\,p}$ to be the list of r_i^* such that r_i has p in its scope. Finally, we define $\bar{r}^{all\,p}$ (resp. $\bar{r}^{*all\,p}$) to be the list of all r_i (resp. r_i^*) in $\bar{r}^{f\,p}$ or in $\bar{r}^{g\,p}$ (resp. in $\bar{r}^{*f\,p}$ or in $\bar{r}^{*g\,p}$).

In many places where we use these variable sequences, it is possible to determine which procedure p is involved from the context. In such situations, we drop explicit mention of p and write, for example, $\bar{r}^{*f}$ as an abbreviation for $\bar{r}^{*f\,p}$.

At various places in the proof, we will have occasion to refer to individual variables in one of these sequences. We adopt the following notation: r^{f_i} stands for the i^{th} element of $\bar{r}^f$. Elements of other sequences are defined similarly.

Example. In order to help explain the construction of π^*, we will refer to the following example throughout the text. Let π_{ex} be the following program:

```
Begin
    p1(r1, x1) ←
        Begin
            p2(x2) ← Begin r1(x2); r1(x2) End;
            If x1 = 0 Then r1(x1) Else Begin x1 := x1 − 1; p1(p2, x1) End
        End;
    p3(x3) ← x3 := x3 + 1;
    p1(p3, x4)
End.
```

This program is in a form that satisfies the naming assumptions: all declarations have distinct main identifiers, and no formal procedure name appears in more than one formal part in the program. Let us consider some examples of the sequence notation for formal parameters. The procedure p_1 has a formal procedure parameter r_1, so $\bar{r}^{f\,p_1}$ is the sequence $[r_1]$. The declaration of p_1 is not in the scope of any formal procedure, so $\bar{r}^{g\,p_1}$ is the empty sequence. Thus, $\bar{r}^{all\,p_1}$ is $[r_1]$. The declaration of p_2, on the other hand, has no formal procedures but it is in the scope of r_1, so $\bar{r}^{f\,p_2}$ is empty, and $\bar{r}^{g\,p_2}$ and $\bar{r}^{all\,p_2}$ are the sequence $[r_1]$. (End of Example.)

Notation. *In the following presentation, we omit the parameter* b, *which is assumed to appear as the first ground parameter in all procedure calls of the transformed program.*

6. In the following, $\bar{r}$ is the sequence of formal procedure identifiers appearing in the formal part of a declaration with main procedure identifier p_0. The sequence $\bar{x}$ is the sequence of ground variables in the formal part of the declaration of p_0.

(Begin $p_0(\bar{r}, \bar{x})$ ← body; S End)* = Begin $p_0^*(\bar{r}^{*\,all}, \bar{x})$ ← body*; S* End,

where p_0^* is a new declared procedure identifier.

7. Translation of calls on declared procedures. Suppose p_0 is a declared procedure of type τ, $\bar{q}$ is a list of declared and formal procedure identifiers, and $\bar{y}$ is a list of ground variables. The translation introduces a set of new local code variables, $r'_1,\ldots,r'_{|\tau|}$, to hold the codes for the $|\tau|$ procedure parameters in the call.

The definition of the translation depends on the correspondence of the names for the actual and formal procedures in the call on p_0. In this case, the k^{th} actual parameter is q_k. The k^{th} formal parameter of p_0 is the k^{th} procedure identifier in the

formal-list in the declaration of p_0. Using our notation, this is r^{*f_k}, and its code parameter is r^{*f_k}.

$(p_0(\bar{q}, \bar{y}))^* =$

```
Begin
  var r'_1,...,r'_|τ|;        {declare |τ| local code variables}
  r̄' := e_S(r̄*);              {assign a code value to the r̄' variables}
  p*_0(r̄*^all θ, ȳ)
End,
```

where θ is the substitition $[r'_1/r^{*f_1}, \ldots, r'_{|\tau|}/r^{*f_{|\tau|}}]$.

The statement $\bar{r}' := e_S(\bar{r}^*)$ is a shorthand for a program that assigns a code value to each of the variables in $\bar{r}'$, using the $\bar{r}^*$ as inputs. This program depends on the statement S, i.e. the particular call $p_0(\bar{q}, \bar{y})$. We will explain the details of the program below. In the call on p_0^*, note that the substitution θ simply replaces the variables $\bar{r}^{*f}$ (the codes for the formal procedure parameters of p_0) by the variables $\bar{r}'$.

Notation. In the rest of the paper we will use a simplified notation for the statement $p_0^*(\bar{r}^{*all}\theta, \bar{y})$ in the simulation. For ease of understanding, we will write this statement as $p_0^*(\bar{r}', \bar{r}^{*g}; \bar{y})$. Thus the call on p_0^* will have the form $p_0^*(<$code variables for formal parameters$>$, $<$code variables for global parameters$>$; $<$original ground parameters$>$).

Example (cont.) Before describing the details of the calculation of codes, we will give an example of the general form of the translatien. Consider the call S1 = $p_1(p_2, x_1)$, which appears in the body of p_1 in π_{ex}. The translation will be a statement of the form

```
Begin
    var r'_1;
    r'_1 := e_S1(r*_1);
    p*_1(r'_1, x_1)
End.
```

Since $p_1(p_2, x_1)$ is a call on the declared procedure p_1, the translation will call the new procedure p_1^*. The procedure p_1^* is defined to have one code parameter in place of a formal procedure parameter. The translation defines a local variable r'_1 to hold the code

for p_2. The variable r'_1 is assigned a value by the assignment statement $r'_1 := e_{S1}(r^*_1)$. Intuitively, one can see that since the procedure p_2 has r_1 free in it, the calculation of a code for a closed declaration equivalent to p_2 can be done if we are given the code for r_1. That is the reason that the code variable r^*_1 is needed in the calculation. (End of Example.)

We will now present the formal details of the calculation of code values. Each of the local code variables, r'_i, $i = 1,...,|\tau|$, is assigned a value according to one of the following cases:

7a. If the i^{th} actual procedure parameter in the call, q_i, is the <u>formal</u> procedure r_j, then we set r'_i to the code for r_j:

$r'_i := r^*_j$.

Since we want the translation to be an L4 program, we need to make sure that the translation does not introduce global references to any of the ground variables. Let us say that an identifier is <u>bound</u> by a procedure declaration if the identifier is in the formal list of the declaration; we will say an identifier is bound in an instance of a statement if the *innermost* procedure that contains the statement binds the identifier. Thus to make sure that the statement $r'_i := r^*_j$ does not introduce any global references to variables, we must check that r^*_j is bound where this statement occurs.

Note that if r_j is bound in the original call in π, then r^*_j is bound in the translation of the call in π^* for the following reason: Suppose the innermost procedure containing the call is p'. Then if r_j appears in the call, r_j must be in either $\bar{r}^{f\,p'}$ or $\bar{r}^{g\,p'}$, because r_j must be either a parameter of p' or a parameter of some procedure that encloses p'. In either case, r^*_j will be in $\bar{r}^{*all\,p'}$, and so will be bound in the translation.

An example will help to explain this point. In the program π_{ex}, the formal procedure r_1 is bound by procedure p_1, so it can be used throughout the body of p_1. In the translation, the variable r^*_1 will be bound by p^*_1, because r_1 is in $\bar{r}^{f\,p_1}$. Now, consider the procedure p_2, which is nested within p_1. The formal r_1 appears free in the body of p_2. In the translation, r^*_1 is bound by p^*_2 because r_1 is in $\bar{r}^{g\,p_2}$. Thus r^*_1 is bound in the translation where it is needed.

7b. If the i^{th} actual procedure parameter in the call, q_i, is a <u>declared</u> procedure, say p, the simulation sets r'_i to a code for a closed declaration equivalent to p in the "current environment." Intuitively, we can construct this closed declaration in two steps. First, we take the declarations of all the declared procedures that are reachable from the body of p. This collection of declarations can have free <u>formal</u> procedure

names. The second step is to add a closed declaration for each of these free formal procedure names. These declarations are determined by the code variables r_i^*. In effect, we add the declarations encoded by the code variables. Roughly speaking, from all of these declarations we can then form a closed declaration equivalent to p in the "current" procedure environment.

We will now describe precisely how the closed declaration is formed. It is important to recall that we assumed that π has been put into a form such that all declarations have distinct main identifiers, and formal procedure names appear in the formal list of no more than one declaration. Because there can be no clashes of procedure names, all of the declarations in force at the place where a procedure is declared are also in force anywhere in the scope of the procedure.

Now, let us see which of the declarations of π are needed in order to form a closed declaration equivalent to p. If p is a declared procedure name and q is any procedure name, let us say that q is reachable from p by following free procedures for 0 steps if q is free in the body of p, and say that q is reachable from p by following free procedures for k+1 steps if there is some declared procedure p′ that appears free in the body of p, and q is reachable from p′ by following free procedures for k steps. Then we will say that q is reachable from p by following free procedures if q is reachable from p by following free procedures for k steps, for some natural number k. Furthermore, if S is a statement of π, then let us say that q is reachable from S by following free procedures iff q is free in S or q is reachable by following free procedures from some procedure free in S.

For example, in π_{ex}, the procedure r_1 is reachable by following free procedures from p_2, because r_1 is free in the body of p_2. Now, consider the call $p_1(p_2, x_1)$, which appears in the body of the procedure p_1. The procedures reachable from this statement by following free procedures are p_1, p_2, and r_1. The first two are reachable by following free procedures because they are free in the statement. The procedure r_1 is reachable by following free procedures because the declared procedure p_2 is reachable, and because r_1 appears free in the body of p_2.

Now, let us fix a declared procedure p and define some environments related to p in π. Let $E_{declared}$ be the environment consisting of the declaration of p and declarations of all declared procedures reachable from p by following free procedures. Since all procedures declared in π have distinct names, there are no ambiguities in relating the declared procedure names reachable from p to their bodies. Clearly, the only procedures that can be free in $E_{declared}$ are formal procedure names. To form a closed declaration for p, we need to have closed declarations for all of the formal procedure

names free in $E_{declared}$. Then we could define p with a closed declaration of the form

$$p(\bar{r}, \bar{x}) \leftarrow (E_{declared} \cup E_{formal}) \mid p(\bar{r}, \bar{x}),$$

where E_{formal} has closed declarations for each of the formal procedure names free in $E_{declared}$, and $\bar{r}$ is a list of procedure names distinct from the the names in E_{formal}.

In the simulation, the environment E_{formal} will be formed from the code variables. In order to do this, we will use a 1-sided inverse of the encoding function ρ. First, let T be the set of all procedure types mentioned in π, and ρ be an encoding function for Closures(T) in I. From now on, let ρ^{-1}: $\mathbf{N} \rightarrow$ Closures(T) be a function such that $\rho(\rho^{-1}(n)) = n$, for all n in the image of Closures(T) under ρ. In other words, ρ^{-1} is defined for all codes of closed declarations of types mentioned in π. Given a code for a closed declaration d, ρ^{-1} yields a declaration that is semantically equivalent to d.

The environment E_{formal} is essentially $\{\rho^{-1}(r_i^*) \mid r_i \text{ is free in } E_{declared}\}$. However, for technical reasons, the exact description of E_{formal} involves renaming. During the simulation, r_i^* is set to codes for various declarations; because of the way we define the simulation, the main procedure identifier of these declarations is *not* r_i. In general, the code variables are set to codes for declarations with other procedure identifiers in π. In order to define E_{formal} in terms of the values of the code variables, we need to use the renaming operator $\nu_{r_i}(d)$, which takes a declaration d and renames the main procedure identifier to r_i. The exact description of E_{formal} is $\{\nu_{r_i}(\rho^{-1}(r_i^*)) \mid r_i \text{ is free in } E_{declared}\}$.

This completes the syntactic definition of the declaration that must be constructed. Next, let us see how the simulation program can compute a code for the declaration $p(\bar{r}, \bar{x}) \leftarrow (E_{declared} \cup E_{formal})|p(\bar{r}, \bar{x})$. We will use the program $BIND_d$, where d is the declaration $p(\bar{r}, \bar{x}) \leftarrow E_{declared}|p(\bar{r}, \bar{x})$. This BIND program takes codes for the procedure names free in d, and produces a code for a closure. The procedure names free in d are the formal procedure names in E_{formal}. The simulation can compute codes for the declarations in E_{formal} from the code variables $\bar{r}^*$. The simulation must use RENAME programs to rename the declaration in r_i^* to have main procedure identifier r_i.

Example (cont.) Returning to the example, let us again consider the translation of the call $S1 = p_1(p_2, x_1)$ in π_{ex}. The translation had the form

```
Begin
    var r'1;
    r'1 := eS1(r1*);
    p1*(r'1, x1)
End,
```

where the assignment statement $r'_1 := e_{S1}(r^*_1)$ is an abbreviation for a program that sets r'_1 to a code for the procedure p_2 in the current environment. Let us see what this program does. In this case, $E_{declared}$ contains just the declaration of p_2, because no other declared procedures are reachable from p_2. The environment E_{formal} will have a declaration for r_1 because r_1 is free in E_{formal}. Thus, we will use the $BIND_d$ program, where d is the declaration $p_2(x_2) \leftarrow$ Begin $r_1(x_2)$; $r_1(x_2)$ End. Then the translation uses a renaming program to make a declaration with main procedure identifier r_1 from the value of r^*_1. Then the BIND program makes a closed declaration for p_2. Finally, r'_1 is set to the result of the BIND program.

This example illustrates the technical problem of renaming main procedure identifiers. The result of the BIND program is a code for a declaration with main procedure identifier p_2. In the call $p^*_1(r'_1, x_1)$ in the body of p^*_1, the variable r^*_1 thus takes on the value of a code with main procedure identifier p_2. The renaming is thus needed somewhere; we found it simplest to apply renaming just before the values are used. (End of Example.)

This completes the description of how the simulation computes a code for a closed declaration for p. However, in order to show that the program for computing the code is well formed, we need to make sure that all of the code variables needed to compute E_{formal} are bound where their values are used.

Recall that a code variable is only bound in certain procedures in π^*. Previously, we discussed the assignment of code values in the case of a formal procedure appearing as an actual in a call (section 7a). There, we considered simulating a procedure call contained in the body of a procedure, say p′. It was a simple matter to see that if a code variable was used in simulating the procedure call, then, by the definition of π^*, the code variable was bound in the translation of the procedure p′. Now we must make a similar argument for the code variables that are used when a declared procedure appears as the actual in a call.

Suppose the procedure p is an actual parameter in a procedure call and the call is contained in the body of some procedure p′. We will now show that all of the necessary code variables are bound in p′*, the procedure that simulates p′. This can be shown by

considering relationships between the scopes of the procedures of π.[7,8] We begin by observing that scopes have a transitive property.

Lemma 6.1. If a procedure p (resp. a statement S) is in the scope of a procedure p′, and p′ is in the scope of a procedure q, then p (resp. S) is in the scope of q.

Next, we use the transitive property to show a simple relation between reachability and scopes:

Lemma 6.2. If a procedure q is reachable from p by following free procedures, then p is in the scope of q.

Proof. The proof is by induction on the number of steps needed to reach from p to q. The base case is when q is reachable from p by following free procedures for 0 steps. By definition, this means q is free in the declaration of p; then clearly p is in the scope of q. For the induction step, assume that for all p and q, if q is reachable from p by following free procedures for k steps, then p is in the scope of q. Now suppose q is reachable from p by following free procedures for k+1 steps. Then there is some p′ free in p such that q is reachable from p′ in k steps. Clearly, p must be in the scope of p′. By the inductive hypothesis, p′ is in the scope of q. By the transitive property of Lemma 6.1 we can conclude that p is in the scope of q. □

Next, recall that the environment $E_{declared}$ consists of the declaration of a procedure p and all of the declared procedures reachable from p by following free procedures. A formal procedure r is free in $E_{declared}$ iff it is reachable from p by following free procedures. This gives us the following corollary.

Corol. 6.3. Let p be a declared procedure and $E_{declared}$ be the environment defined for p. If r is a formal procedure name free in $E_{declared}$, then p is in the scope of

---

[7]The remainder of this argument may be omitted on the first reading by skipping to the end of Lemma 6.4.

[8]An alternative way of constructing π^* would be to include all of the $\bar{r}^*$ code variables as parameters to each procedure in the simulation. On first examination, it may appear that this would simplify the proof, because we would not have to be concerned with which variables are bound at which places. However, when the full proof is considered, the alternate construction of π^* would simplify this part of the proof at the expense of adding a greater amount of complication in the part of the proof that uses the axioms. That part of the proof depends on the fact that the semantics of a simulation procedure p* is affected only by variables in $\bar{r}^{*all}$ and not by the other code variables. So we purposely restricted the declaration of p* to use only the variables in $\bar{r}^{*all}$ in order tc reduce the complexity of Part 2 of the proof. With the alternate construction, an argument similar to Lemmas 6.1 – 6.4 would still be needed, and the result would be much less convenient to apply in Part 2.

r. Equivalently, if r is free in $E_{declared}$, then r is in $\bar{r}^{g\,p}$.

We can now complete our discussion of the code variables needed for the simulation of a procedure call in which p appears as an actual parameter. Let p′ be the innermost procedure containing this (instance of) the call. That is, p′ contains the call and every other procedure of π that contains the (instance of) the call also contains the declaration of p′. We have seen that the simulation requires all the code variables r_i^* such that r_i is free in the declaration of p or reachable from p by following free procedures. We can now show that all of these code variables are bound by p′*.

Lemma 6.4. If r is a formal procedure name free in $E_{declared}$, then r* is bound by p′*.

Proof. We must consider two cases. If p is free in the declaration of p′, then p′ is in the scope of p. By Lemma 6.3 and the transitive property, p′ is in the scope of r. But then, by definition, r is in $\bar{r}^{g\,p'}$, and so r* is bound by p′*. Now, suppose on the other hand, that p is not free in the declaration of p′. In this case, p must be declared within p′. Furthermore, since p′ is the innermost procedure containing the procedure call with p as an actual parameter, p′ must also be the innermost procedure containing the declaration of p. Similarly, one can see that if p″ is any declared procedure reachable from p by following free procedures, then either the declaration of p″ is within p′, and p′ is the innermost procedure containing this declaration, or p′ is in the scope of p″. It follows from this that if r is a formal procedure reachable from p by following free procedures, then either r is bound by p′ or p′ is in the scope of r. If r is bound by p′, then r* is bound by p′*, and as before, if p′ is in the scope of r, then r* is also bound by p′*. □

We will now summarize the results of the construction of 7b. Sections 7a and 7b describe the construction of a program that sets the local code variables $\bar{r}'$. We abbreviated this entire program by the statement $\bar{r}' := e_S(\bar{r}^*)$. In section 7b, we have been describing the program that assigns a value to r_i' in the case that the i^{th} actual parameter in a procedure call is a declared procedure. We can abbreviate this program by the statement $r_i' := e_i(\bar{r}^*)$. Actually, the only $\bar{r}^*$ variables that are used in the statement are those that correspond to formal procedures that are free in $E_{declared}$. We showed in Lemma 6.3 that all of these procedures are in $\bar{r}^{g\,p}$, so the corresponding code variables are all in $\bar{r}^{*g\,p}$.

Note that the program $r_i' := e_i(\bar{r}^*)$ is guaranteed to terminate (for some value of b -- not shown), setting r_i' to a code for a closed declaration of the same type as p,

whenever each variable r_i^* in $\bar{r}^{*g\ p}$ is initially set to a domain value such that $\phi(r_i^*)$ is defined and is a natural number code for a closed declaration of the same type as r_i. We will refer back to this fact about definedness in Part 2.

This completes the translation of calls on declared procedures.

The translation of calls on formal procedures is similar to the case of calls on declared procedures. The main difference is that instead of calling a procedure p*, the translation calls an interpreter program. The translation for $S = r_i(\bar{q}, \bar{y})$ is as follows:

8. $(r_i(\bar{q}, \bar{y}))^* =$

```
Begin
  var r'_1,...,r'_|τ|;
  r̄' := e_S(r̄*);        {assign a code value to each of the r̄'}
  <r_i*>(<r̄'>, ȳ)
End,
```

where τ is the type of r_i, $r'_1,...,r'_{|\tau|}$ are new code variables, and $\langle r_i^* \rangle(\langle \bar{r}' \rangle, \bar{y})$ is a simulation program which uses programs from Lemma 4 (explained below).

The essential difference between the calls $p_i(\bar{q}, \bar{y})$ and $r_i(\bar{q}, \bar{y})$ is that in the first call, the main procedure p_i has a fixed, known body in the context of π, while in the second one, r_i ranges over a possibly infinite set of procedures. In the translation of $p_i(\bar{q}, \bar{y})$, we introduced a fixed procedure p_i^*, which could simulate calls to p_i with different possible actual parameters. We did this by defining p_i^* to have a body that was structurally similar to the body of p_i, but which used code variables instead of procedures. In order to translate calls to r_i, we use an interpreter program to handle the possible infinite range of the main procedure.

We define $\langle r_i^* \rangle(\langle \bar{r}' \rangle, \bar{y})$ to be a program with free code variables r_i^*, $\bar{r}'$, and additional free ground variables $\bar{y}$. Intuitively, $\langle r_i^* \rangle$ can be thought of as "the procedure encoded by r_i^*," and $\langle r_i^* \rangle(\langle \bar{r}' \rangle, \bar{y})$ can be thought of as a call of the procedure encoded by r_i^*, with the actual procedure parameters encoded by $\bar{r}'$, and the actual ground parameters $\bar{y}$. Recall that Lemma 4 showed the existence of a program INTERP_τ, defined purely over dom(I), that computes the interpreter relation ι_τ in the sense that when it is started with domain values encoding a main procedure, a list of codes for procedure parameters to the main procedure, and a list of ground parameters to the main procedure, it sets the list of ground parameters to the same values that the unencoded procedure call would.

As defined in Lemma 4, the program INTERP_τ requires all of its code parameters to be codes for procedures with distinct main procedure names. In constructing π^*, it is necessary to use the ν programs at some point to rename the main procedure identifiers in declarations. For example, the environment binding operation is defined to form a declaration that has the same main procedure identifier as the original declaration. Consider a procedure call such as $r_i(r_j, r_k)$, where r_j and r_k have the same type. A program can reach an environment where r_j and r_k correspond to one declaration p:d in two different environments, E|p:d and E′|p:d. In the simulation, it is necessary to use renaming in the interpreter program to prevent these two declarations from clashing. For convenience, we will use a version of the interpreter program that always renames it code parameters to be declarations with distinct main procedure names. This is simply a convenience to make it unnecessary to maintain distinct names at all times.

For the rest of the proof, we will take the program $\langle u\rangle(\langle\bar{v}\rangle, \bar{y})$ to be a program that first renames its code arguments u and $\bar{v}$, to have distinct names, sets the result of the renaming into u′ and $\bar{v}'$, and then invokes $\text{INTERP}_\pi(u', \bar{v}', \bar{y})$. In each context where we use this notation, the code u will be associated with some intended type τ, and the interpreter program actually used will be the one for that type. For instance, in the above translation of calls to r_i, τ is the type of r_i.

The first code variable in $\langle r_i^*\rangle(\langle\bar{r}'\rangle, \bar{y})$ is the code for r_i. The other code variables $\bar{r}'$ specify closures for the $|\tau|$ formal parameters to r_i. Since all of the closures are closed declarations, this information is sufficient to interpret the call with the ground variables $\bar{y}$.

In defining $(p(\bar{q}, \bar{y}))^*$ it was necessary to carry along codes for global formal procedures because of the possibility that free procedures could be referenced by one of the declared procedures. This is not needed in this case, because instead of calling one of the procedures p_i^*, the translation calls INTERP_τ with closures for all the formal procedures.

The setting of the code variables $\bar{r}'$ is the same as the previous case. There are two cases to consider: an actual parameter can be either a formal procedure or a declared procedure. The values are assigned in the same way.

This completes the definition of S*, for each statement S. Note that if π is an L4 program, then the declarations in π^* have no free occurrences of program variables and no free occurrences of procedure identifiers except for the procedures p*. There are no free references to formal procedures. This means that the declarations in π^* can be moved from inner scopes to the outermost scope without changing their meaning.

Example. (cont.) The complete translation of the example program π_{ex} is:

```
Begin
    p1*(r1*, x1) ←
        Begin
            p2*(r1*, x2) ← Begin <r1*>(x2); <r1*>(x2) End;
            If x1 = 0 Then <r1*>(x1)
                      Else Begin
                               x1 := x1 − 1;
                               Begin
                                   var r1';
                                   r1' := eS1(r1*);
                                   p1*(r1', x1)
                               End
                           End
        End;
    p3*(x3) ← x3 := x3+1;
    Begin
        var r1';
        r1' := e2;
        p1*(r1', x4)
    End
End.
```

6.5. Semantic properties of the simulation

In this section we develop some results about the semantics of the simulation program. These results will be used in the completeness proof.

We summarize the results of the construction of π^* in the following lemma, which states that the semantics of a statement S in the environment $E_{formal} \cup E_{declared}$ is essentially the same as the semantics of $(E_{declared}|S)^*$, provided the code variables $\bar{r}^*$ are initially set to an encoding of the formal procedures E_{formal}.

Lemma 7.1. Let S be a statement of π, and $E_{declared}$ be the environment of all declared procedures reachable from S by following free procedures. Let $\bar{r}$ be the sequence of formal procedure names free in either S or $E_{declared}$. Let $E_{formal} = \{r_1{:}d_1,\ldots,r_n{:}d_n\}$ be an environment that assigns closed declarations of the correct types to each name in $\bar{r}$. We will use the function ϕ:NUM $\rightarrow$ Codes(T) from Lemma 4, that maps elements of NUM (i.e. k-tuples of domain elements) into the natural numbers that the domain elements represent. Let $\bar{v}^n$ be a sequence of n k-tuples of domain elements such that v_i represents the code for $r_i{:}d_i$, that is $\phi(v_i) = \rho(r_i{:}d_i)$, for i=1,...,n. Finally,

let $\bar{r}^*$ be the sequence of code variables corresponding to the sequence of procedure names $\bar{r}$.

Under these assumptions,

$$(\sigma[\bar{x} \leftarrow \bar{u}], \sigma[\bar{x} \leftarrow \bar{u}']) \in \mathcal{M}_I((E_{formal} \cup E_{declared})|S)$$
iff
$$\exists z\ ((\sigma[\bar{x} \leftarrow \bar{u}, b \leftarrow z, \bar{r}^* \leftarrow \bar{v}], \sigma[\bar{x} \leftarrow \bar{u}', b \leftarrow z, \bar{r}^* \leftarrow \bar{v}]) \in \mathcal{M}_I((E_{declared}|S)^*)).$$

For the full program π, which has no free procedures, Lemma 7.1 has a simpler form. We need only this special case of the lemma for the completeness proof. In order to state the simpler form of the lemma, we will define what it means to existentally quantify over a variable on a relation on $val_I \times val_I$. If $\mathcal{R}$ is a subset of $val_I \times val_I$ and x is variable, then $\exists x\mathcal{R}$ is the relation defined by $(\sigma, \sigma') \in \exists x\mathcal{R}$ iff $\exists u((\sigma[x \leftarrow u], \sigma'[x \leftarrow u]) \in \mathcal{R})$.

Lemma 7.2. The semantics of π is essentially the same as the semantics of π^*, that is, $\mathcal{M}_I(\pi) = \exists b \mathcal{M}_I(\pi^*)$.

Proof. This is simply a restatement of 7.1 in the case that there are no free procedure names. □

Notation. In the remainder of the proof, we will frequently need to express strongest postconditions of statements in π^*. Since π^* has no procedures passed as parameters, the only procedure names that can appear free in a statement are the procedures declared in π^*, the p_i^*. For a statement S^* of π^*, we will write SP [S^*; Q] to denote the strongest postcondition of S^* with respect to Q in the procedure environment of π^*. To state this precisely, let $D\pi^*$ be the set of all declarations occurring in π^*. Then SP [S^*; Q] is an abbreviation for SP [$D\pi^*|S^*$; Q].

In the proof of the completeness theorem, we will need a special property of the simulation program, having to do with the input parameter b and the simulation of arithmetic. In bounded interpretations, the role of b is to supply a pair of domain values that can be used to simulate a fixed, finite amount of arithmetic using an encoding such as the one discussed in Part 1. In unbounded interpretations, b supplies starting values that are used to generate other reachable domain values in order to simulate arithmetic. We now show that the semantics of the statements in the simulation have a certain special property involving the variable b.

For $\mathcal{R}_1$ and $\mathcal{R}_2$ subsets of $\text{val}_I \times \text{val}_I$, let $\mathcal{R}_1 \circ \mathcal{R}_2$ denote the composition of the relations $\mathcal{R}_1$, $\mathcal{R}_2$, that is, $(\sigma, \sigma') \in \mathcal{R}_1 \circ \mathcal{R}_2$ iff for some σ'', $(\sigma, \sigma'') \in \mathcal{R}_1$ and $(\sigma'', \sigma') \in \mathcal{R}_2$.

For *any* two programs in the language $\pi 1$ and $\pi 2$, we have $\mathcal{M}_I(\pi 1; \pi 2) = \mathcal{M}_I(\pi 1) \circ \mathcal{M}_I(\pi 2)$. A consequence is property SP 4 of strongest postconditions: SP [($\pi 1$; $\pi 2$); Q] $\equiv$ SP [$\pi 2$; SP [$\pi 1$; Q]]. We now show that the statements that appear in the simulation program enjoy a special, additional property with respect to the variable b. If S1 and S2 are statements in the simulation, then $\exists b \mathcal{M}_{I,D\pi^*}(S1;S2) = \exists b \mathcal{M}_{I,D\pi^*}(S1) \circ \exists b \mathcal{M}_{I,D\pi^*}(S2)$. This says that the following are equivalent: (1) there is an assignment of a domain value u to the variable b such that the computation of S1;S2 can start in a valuation $\sigma[b \leftarrow u]$ and stop in the valuation $\sigma''[b \leftarrow u]$; (2) there are domain values u1 and u2 of b such that S1 can start in a valuation $\sigma[b \leftarrow u1]$ and reach a valuation $\sigma'[b \leftarrow u1]$, and S2 can start in $\sigma'[b \leftarrow u2]$ and stop in valuation $\sigma''[b \leftarrow u2]$. Note that the semantics of the statements S1,S2 is in the environment $D\pi^*$, which consists of all the procedure declarations in π^*.

Lemma 8. If S1 and S2 are statements in the simulation, then $\exists b \mathcal{M}_{I,D\pi^*}(S1;S2) = \exists b \mathcal{M}_{I,D\pi^*}(S1) \circ \exists b \mathcal{M}_{I,D\pi^*}(S2)$.

Proof. Intuitively, if a statement in the simulation, when started in an initial valuation σ with b set to a certain value u, can halt in a final valuation σ', then for all u′ that are "at least as large as" u, when the statement is started in valuation $\sigma[b \leftarrow u']$ it can halt in valuation $\sigma'[b \leftarrow u']$. What is meant by u′ being at least as large as u is that u′ represents a larger natural number than u does, that is, $\phi(u') \geq \phi(u)$.

For any values u1 and u2 of b such that the statement S1 starts in valuation $\sigma[b \leftarrow u1]$ and stops in valuation $\sigma'[b \leftarrow u1]$, and S2 starts in $\sigma'[b \leftarrow u2]$ and stops in $\sigma''[b \leftarrow u2]$, there is a value of u of b such that S1;S2 starts in valuation $\sigma[b \leftarrow u]$ and halts in $\sigma''[b \leftarrow u]$. In essence, $\phi(u)$ is $\max(\phi(u1),\phi(u2))$.

Thus $\exists b \mathcal{M}_{I,D\pi^*}(S1) \circ \exists b \mathcal{M}_{I,D\pi^*}(S2) \subseteq \exists b \mathcal{M}_{I,D\pi^*}(S1;S2)$.

In the other direction, it is clear that any pair of valuations in $\exists b \mathcal{M}_{I,D\pi^*}(S1;S2)$ must also be in $\exists b \mathcal{M}_{I,D\pi^*}(S1) \circ \exists b \mathcal{M}_{I,D\pi^*}(S2)$, because both S1 and S2 terminate for the same value of b in $\exists b \mathcal{M}_{I,D\pi^*}(S1;S2)$. □

In the completeness proof, we will use this lemma in reasoning about strongest postconditions of the statements in the simulation. The following corollary gives the form of the lemma needed for the proof.

Cor. $\exists b$ SP [(S1; S2); Q] $\equiv$ $\exists b$ SP [S2; $\exists b$ SP [S1; Q]] (provided b does not

appear free in Q).

Proof. From Lemma 8 and the definition of strongest postcondition. □

Finally, we conclude this section by mentioning some facts about the program $r' := e(r^*)$ that will be used in the completeness proof.

Lemma 9. Consider a procedure call $S = q_0(\bar{q}, \bar{z})$, on an arbitrary (either declared or formal) procedure q_0 of type τ. Assume $\bar{r}' := e_S(\bar{r}^*)$ is the program that computes the codes $\bar{r}'$ in $(q_0(\bar{q}, \bar{z}))^*$, and C is a first-order formula that expresses $\exists b\ SP[\bar{r}' := e_S(\bar{r}^*);\ W(\bar{r}^*)]$. Further, assume that the only free variables of C are $\bar{r}^*$ and $\bar{r}'$. Then C has properties 9.1 - 9.4. The intuition behind these properties is given in the proof of the lemma.

9.1. $I \models C \supset W(\bar{r}') \wedge W(\bar{r}^*)$.

9.2. $I \models (\exists \bar{r}'\ C) \equiv W(\bar{r}^*)$.

9.3. If q_k, $1 \leq k \leq |\tau|$, is a <u>formal</u> procedure r_h, then $I \models C \supset r'_k = r^*_h$.

9.4a. If q_k, $1 \leq k \leq |\tau|$, is a <u>declared</u> procedure p_h, and σ, σ' are valuations such that $I,\sigma \models C \wedge W(u)$, then

$$(\sigma, \sigma') \in \exists b M_{I,D\pi^*}(\langle r'_k \rangle(\langle \bar{u} \rangle, \bar{x}))$$
iff
$$(\sigma, \sigma') \in \exists b M_{I,D\pi^*}(p^*_h(\bar{u}, \bar{r}^{*g}; \bar{x})).$$

Proof. For 9.1, observe that none of the variables in $\bar{r}^*$ is changed by the program in C. Thus from the fact that b is not free in $W(\bar{r}^*)$ and from SP 1, we have

$$\begin{aligned} I \models C \wedge W(\bar{r}^*) &\equiv \exists b(SP[\bar{r}' := e_S(\bar{r}^*);\ W(\bar{r}^*)]) \wedge W(\bar{r}^*) \\ &\equiv \exists b(SP[\bar{r}' := e_S(\bar{r}^*);\ W(\bar{r}^*)] \wedge W(\bar{r}^*)) \\ &\equiv \exists b\ SP[\bar{r}' := e_S(\bar{r}^*);\ W(\bar{r}^*)] \\ &\equiv C. \end{aligned}$$

Hence, $I \models C \supset W(\bar{r}^*)$. By the construction of $\bar{r}' := e_S(\bar{r}^*)$, for each valuation σ satisfying $W(\bar{r}^*)$, there is a value v of b such that the program $\bar{r}' := e_S(\bar{r}^*)$, when it is started in the valuation $\sigma[b \leftarrow v]$, halts setting $\bar{r}'$ to values that satisfy $W(\bar{r}')$. This shows that 9.1 holds.

For 9.2, we use the fact that the only free variables of C are $\bar{r}^*$ and $\bar{r}'$. By 9.1, $I \models \exists \bar{r}'\ C \supset W(\bar{r}^*)$. As in 9.1, we use the fact that for each valuation σ satisfying $W(\bar{r}^*)$,

there is a value v of b such that the program $\bar{r}' := e_S(\bar{r}^*)$ always halts when it is started in the valuation $\sigma[b \leftarrow v]$. From this it follows that $I \models W(\bar{r}^*) \supset \exists \bar{r}'\ C$.

The remaining two properties of C relate to the particular codes assigned by the program, as described in steps 7a and 7b of the definition of S*. If q_k is a formal procedure r_h, then r'_k is assigned the value r^*_h. This gives 9.3

If q_k is a declared procedure p_h, then r'_k is set to a code for the procedure p_h in the environment of free formal procedures encoded by $\bar{r}^{*g}$. Thus when started in a valuation satisfying $C \wedge W(\bar{u})$, the interpreter program $\langle r'_k \rangle(\langle \bar{u} \rangle, \bar{x})$ has essentially the same semantics (the same semantics except for b) as the call $p^*_h(\bar{u}, \bar{r}^{*g}; \bar{x})$ which simulates p_h on formal procedures encoded by $\bar{u}$ and global procedures encoded by $\bar{r}^{*g}$. In the completeness proof, we will use the following statement of this fact in terms of strongest postconditions:

9.4b. If q_k, $1 \leq k \leq |\tau|$, is a declared procedure p_h, then

$$I \models C \supset (\exists b\ SP[\langle r'_k \rangle(\langle \bar{u} \rangle, \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u})] \equiv \exists b\ SP[p^*_h(\bar{u}, \bar{r}^{*g}; \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u})]).$$

□

7. Provability in the Axiom System

Notation. We implicitly associate a procedure type with each code variable. If r_i has type τ_i in π, then we associate the type τ_i with r_i^*. Similarly, each code variable in a formula will correspond to some procedure variable and hence to some type. In Lemma 5, we showed that the relation $WELL_\tau$, which is true for $\bar{x} \in dom(I)^k$ provided $\bar{x}$ represents a code for a declaration in closures(D) of type τ, is expressible by a first-order formula, $W_\tau(\bar{x})$. For ease of notation, we now adopt the following convention: If r_i^* is a code variable, then $W(r_i^*)$ is an abbreviation for the first-order formula $W_\tau(r_i^*)$, where τ is the type implicitly associated with r_i^*. For a vector of code variables, such as $\bar{r}^{*n}$, $W(\bar{r}^{*n})$ is an abbreviation for $W(r_1^*) \wedge \ldots \wedge W(r_n^*)$. These abbreviations will be used throughout Part 2.

7.1. The Main Lemma

We are now ready to formulate the inductive hypothesis for the proof of the Completeness Theorem. Much of the work of Part 1 has been to permit us to define this inductive hypothesis. It will have approximately the form

$$Th(I) \vdash H_S \rightarrow \{\bar{x}=\bar{x}0\}\ S\ \{\exists b\ SP[S^*;\ \bar{x}=\bar{x}0]\},$$

where S is an arbitrary statement of π, and H_S is a formula that expresses assumptions about the free procedures of S. We will prove the hypothesis by the usual induction on the structure of S. The assumption H_S will depend on the procedure environment of S. In particular, for the full program π, we will define H_π to be the formula True. After we have proved the inductive hypothesis, we will use the fact that H_π = True to deduce

$$Th(I) \vdash \{\bar{x}=\bar{x}0\}\ \pi\ \{\exists b\ SP[\pi^*;\ \bar{x}=\bar{x}0]\}.$$

But since $\exists b\ SP[\pi^*;\ \bar{x}=\bar{x}0] \equiv SP[\pi;\ \bar{x}=\bar{x}0]$, we will have shown

$$Th(I) \vdash \{\bar{x}=\bar{x}0\}\ \pi\ \{SP[\pi;\ \bar{x}=\bar{x}0]\},$$

from which the Completeness Theorem directly follows.

Let us now return to the inductive hypothesis and attempt to gain an intuitive understanding. We will make use of the relation "less defined than," between two interpreted programs or procedures. For two programs π_0 and π_1, let us say that π_0 is less defined than π_1 iff $M_I(\pi_0) \subseteq M_I(\pi_1)$. Because of possible nondeterminism, this amounts to saying that for every computation of π_0 starting in a valuation σ and halting with a valuation σ', there is a halting computation of π_1 with the same initial and final valuations. We extend this definition to procedures with only ground parameters in the obvious way: If E is an environment that gives meaning to two

procedures r_0 and r_1 of the same type with only ground parameters, then we will say r_0 is less defined than r_1 in E and I iff $E|r_0(\bar{x})$ is less defined than $E|r_1(\bar{x})$ in I. The generalization to higher types will be given when we return to the formal presentation.

Observe that if π_0 is less defined than π_1, then π_0 satisfies any pca that π_1 does. That is, for any first-order formulas U and V, $I \models \{U\}\ \pi_1\ \{V\}$ implies $I \models \{U\}\ \pi_0\ \{V\}$.

Let us now explain the hypothesis H_S in more detail. In order to carry out the structural induction argument in the proof of the main lemma, we will define H_S to be an assertion about all of the procedures that have S in their scope. Since the formula depends on scopes, H_S is actually defined for each instance of the statement S in π. For any instance of a statement S, the following inclusions hold: {procedures free in S} $\subseteq$ {procedures reachable from S by following free procedure names} $\subseteq$ {procedures that have S in their scope}. We define H_S to be an assumption about the last of these sets in order to have the strongest possible assumption about free procedures throughout the structural induction.[9]

The formula H_S expresses a relationship between the procedure identifiers and the code variables $\bar{r}^*$. The form of H_S is a conjunction of assertions, one conjunct for each of the procedure identifiers that have S in their scope. For the moment, let us restrict attention to the case that all procedures have ground parameters only; we will give the general case of procedures with higher types when we return to the formal presentation. For a formal procedure r_i that has s in its scope, H_S will have a conjunct $H(r_i, r_i^*)$, which will say that $r_i(\bar{x})$ is less defined than $(r_i(\bar{x}))^*$. Recall that in the simulation, $(r_i(\bar{x}))^*$ is defined to be a call $\langle r_i^* \rangle(\bar{x})$ on the procedure encoded by the variable r_i^*. Formally, $I,E,\varsigma \models H(r_i, r_i^*)$ iff $E|r_i(\bar{x})$ is less defined than $\langle \varsigma(r_i^*) \rangle(\bar{x})$, where the latter expression denotes a program that calls the procedure encoded by r_i^*. Intuitively, one may think of the conjunct in this case as expressing the relation "r_i is less defined than $\langle r_i^* \rangle$."

In case a declared procedure p has S in its scope, then H_S will have a conjunct that expresses the relation that $p(\bar{x})$ is less defined than $(p(\bar{x}))^*$. Since the simulation program $(p(\bar{x}))^*$ depends on the code variables $\bar{r}^{*g}$, H_S formally expresses a relation between the procedure variable p and the environment variables $\bar{r}^{*g}$.

---

[9]The completeness proof could also be formulated using a weaker hypothesis H_S involving only the procedures reachable from S by following free procedures. The approach taken here of using the strongest possible assumption gives a slightly simpler proof.

We can summarize the last two paragraphs by saying that if $\bar{q}$ is list of arbitrary procedure identifiers that have S in their scope, then H_S expresses the relation that $q_i(\bar{x})$ is less defined than $(q_i(\bar{x}))^*$, for all q_i in the list.

Recall that the inductive hypothesis has the form

$$\mathrm{Th(I)} \vdash H_S \rightarrow \{\bar{x}{=}\bar{x}0\}\ S\ \{\exists b\ SP[S^*;\ \bar{x}{=}\bar{x}0]\}.$$

We now explain the intuition behind the pca on the right of the arrow. First, let π_0 and π_1 be any two programs, and consider the pca $\{\bar{x}{=}\bar{x}0\}\ \pi_0\ \{SP[\pi_1;\ \bar{x}{=}\bar{x}0]\}$, where $\bar{x}$ includes of all the ground variables free in either program, and $\bar{x}0$ is a list of new variables not free in either program. It is apparent from the definitions of truth of a pca and strongest postcondition that this pca is true iff π_0 is less defined than π_1. Thus $I,E,\xi \models \{\bar{x}{=}\bar{x}0\}\ S\ \{\exists b\ SP[S^*;\ \bar{x}{=}\bar{x}0]\}$ if $E|S$ is less defined than S^*. Intuitively, since H_S will be defined to mean "$q(\bar{y})$ is less defined than $(q(\bar{y}))^*$," we may read the inductive hypothesis in this case as saying, "*It is provable in the axiom system* that whenever $q(\bar{y})$ is less defined than $(q(\bar{y}))^*$, then S is less defined than S^*."

We now return to the formal presentation.

Definition. For each procedure type τ, we now define a formula with a single free procedure name q of type τ and a single free environment variable v. Intuitively, this formula will assert that if v is a code for a closed procedure of type τ, then q is less defined than <v> (the procedure encoded by v). For notational convenience, we will introduce an infix notation and write $q \ll_\tau v$ as an abbreviation for the formula. Since the type τ in the formula $q \ll_\tau v$ is the type of q, we will drop the subscript and simply write $q \ll v$ when the type of q can be determined from the context.

If τ does not contain any other procedure types, i.e. $|\tau| = 0$, then $q \ll v$ is the pca

$$q \ll v \stackrel{def}{=} \{\bar{x} = \bar{x}0 \wedge W(v)\}\ q(\bar{x})\ \{\exists b\ SP[\langle v\rangle(\bar{x});\ \bar{x} = \bar{x}0 \wedge W(v)]\}, \quad (\text{if } |\tau| = 0).$$

This formula is satisfied by a procedure q and domain value v, under the following two conditions: (1) v is a code (for a declaration of type τ) and any valuation reachable by starting with $\bar{x} = \bar{x}0$ and executing $q(\bar{x})$ must satisfy the strongest postcondition of the simulated call $\langle v\rangle(\bar{x})$, or (2) v is not a code. Intuitively, in case 1, the procedure q is less defined than the procedure encoded by v.

We proceed by induction on types. If τ is $(\tau_1,...,\tau_{|\tau|}, \mathrm{var}^{\|\tau\|})$, then $q \ll v$ is

$$q \ll v \overset{def}{=}$$
$$\forall \bar{g}^{|\tau|}\, \forall \bar{u}^{|\tau|} \Big(\Big(\bigwedge_{i=1}^{|\tau|} g_i \ll u_i \Big) \rightarrow$$
$$\{\bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(v)\}$$
$$q(\bar{g}, \bar{x})$$
$$\{\exists b\ SP\,[\langle v\rangle(\langle \bar{u}\rangle, \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(v)]\}\Big),$$
$$(\text{if } |\tau| \neq 0)$$

where $\bar{g}$ is a sequence of procedure variables with g_i having type τ_i, and $\bar{u}$ is a sequence of environment variables. Intuitively, this formula says that a higher-typed procedure q is less defined than <v> iff for all procedures $\bar{g}$ having the right types and codes $\bar{u}$, whenever all of the g_i are less defined than $\langle u_i\rangle$, then $q(\bar{g}, \bar{x})$ is less defined than $\langle v\rangle(\langle \bar{u}\rangle, \bar{x})$.

We now use the formula $\ll$ to define the conjunct of the assumption H_S. If r_i is a formal procedure identifier of type τ_i in π, then we define R_i to be $\ll$ with appropriate free variables substituted; i.e.

$$R_i(r_i, r_i^*) \overset{def}{=} r_i \ll r_i^*.$$

Thus R_i simply asserts that the formal procedure r_i is less defined than $\langle r_i^*\rangle$. As a convenience for the reader, when we first define formulas that will appear later in the proof, we will list the formula's free variables in parenthesis.

If p_i is a declared procedure identifier of type τ_i in π, then we define P_i to be a formula with the following free variables: the procedure identifier p_i appears free, and of the full set of environment variables $\bar{r}^*$, only those in $\bar{r}^{*g\,p_i}$ appear free.

$$P_i(p_i, \bar{r}^{*g\,p_i}) \overset{def}{=}$$
$$\forall \bar{g}^{|\tau|}\, \forall \bar{u}^{|\tau|} \Big(\Big(\bigwedge_{k=1}^{|\tau|} g_k \ll u_k \Big) \rightarrow$$
$$\{\bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(\bar{r}^{*g})\}$$
$$p_i(\bar{g}, \bar{x})$$
$$\{\exists b\ SP\,[p_i^*(\bar{u}, \bar{r}^{*g}; \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(\bar{r}^{*g})]\}\Big),$$

where $\bar{g}$ is a list of procedure variables and $\bar{u}$ is a list of environment variables. (Recall that $SP\,[p_i^*(..);\ Q]$ is an abbreviation for the strongest postcondition of a program,

$SP\,[D\pi^*|p_i^*(..);\ Q]$, as defined before Lemma 8.)

If H_1, H_2, ..., is a sequence of formulas and $\Phi(i)$ is a condition that is true for a finite number of values of i, then we define $(\bigwedge_i H_i \mid \Phi(i))$ to be the conjunction of the H_i s.t. $\Phi(i)$ is true. If $\Phi(i)$ is never true, then $(\bigwedge_i H_i \mid \Phi(i))$ is defined to be the formula True.

We can now define the assumption H_S. If S is a statement of π, then H_S is $P^S \wedge R^S$, where

$$P^S(\bar{p}, \bar{r}^*) \stackrel{def}{=} (\bigwedge_i P_i \mid S \text{ is in the scope of } p_i),$$

$$R^S(\bar{r}, \bar{r}^*) \stackrel{def}{=} (\bigwedge_i R_i \mid S \text{ is in the scope of } r_i).$$

Observe that because the full program π has no free procedures, it is not in the scope of any procedure. Hence, $P^\pi = R^\pi = \text{True}$.

The Completeness Theorem follows as a consequence of standard techniques from the following lemma:

Main Lemma. If π is a program in L4, I is an expressive interpretation, S is an instance of a statement of π, and x is the sequence of program variables free in S, then

$$Th(I) \vdash P^S \wedge R^S \rightarrow \{\bar{x} = \bar{x}0 \wedge W(\bar{r}^*)\}\ S\ \{\exists b\ SP\,[S^*;\ \bar{x} = \bar{x}0 \wedge W(\bar{r}^*)]\}.$$

Proof of Main Lemma. The proof proceeds by induction on the structure of S. The Recursion Rule is used to discharge the assumptions P^S and R^S. The most important cases in the proof are the base cases for procedure call statements. P_i is a "most general formula" for calls on the declared procedure p_i -- it describes the semantics of a call on p_i for any set of parameters. Similarly, R_i is a most general formula for calls on the formal procedure r_i. When reasoning about a particular call, we instantiate the universally quantified variables in the most general formula and then apply derived axiom 13 to logically weaken the general formula. In Section 4.2.4, we defined $C \rightsquigarrow H$, where C is a first-order formula whose only free variables are environment variables and H is an arbitrary formula, to be a certain formula that is semantically equivalent to $C \rightarrow H$. The meaning of such a formula is "If the environment valuation satisfies C, then H holds." Note that $H \rightarrow (C \rightsquigarrow H)$ is semantically valid because of the equivalence of $C \rightsquigarrow H$ and $C \rightarrow H$. We showed that

$$\vdash H \rightarrow (C \rightsquigarrow H),$$

and we use this formula as derived axiom 13. In the completeness proof, this axiom is used to weaken the most general formula for reasoning about a particular call. For example, when reasoning about a call on p_i, we would deduce

$$\mathrm{Th(I)} \vdash P_i \rightarrow (C \rightsquigarrow P_i'),$$

where P_i' is an instantiation of P_i. Roughly speaking, the first-order formula C is chosen to assert that the environment variables $\bar{r}^{*f\ p_i}$ are set to codes for the actual parameters of the particular call. With such a choice of C, the formula for the Main Lemma can then be deduced using other conventional reasoning. Reasoning about calls on a formal procedure r_i is similar and involves specializing the formula R_i by instantiating it and using the derived axiom.

Notation. In the remainder of the proof we will be discussing provability with respect to Th(I). To avoid repeating this, we adopt the convention for the remainder of the proof that H1 $\vdash$ H2 means Th(I), H1 $\vdash$ H2, except if we specifically mention that this convention is not to be applied.

7.2. Proof of the Main Lemma for a call on a declared procedure

Consider a statement $S = p_i(\bar{q}, \bar{z})$, where p_i is a declared procedure of type τ, $\tau = (\tau_1,\ldots,\tau_{|\tau|}, \mathrm{var}^{||\tau||})$. Each of the q_j is either a declared or formal procedure identifier of π with type τ_j, and $\bar{z}$ is a sequence of ground variables.

First we will give a brief overview of the proof and then the details. By assumption, P_i, the most general formula for calls on p_i, is a conjunct of P^S in the hypothesis. By definition, P_i has the form

$$\forall \bar{g}^{|\tau|}\ \forall \bar{u}^{|\tau|}\ (\ (\ \bigwedge_{k=1}^{|\tau|} g_k \ll u_k)\ \rightarrow\ \mathrm{pc}[\ p_i(\bar{g}, \bar{x})\ ]\ ),$$

where pc is a pca involving the call $p_i(\bar{g}, \bar{x})$. In order to use this formula to reason about the call $p_i(\bar{q}, \bar{z})$, we will first instantiate the bound variables $\bar{g}$ and $\bar{u}$. We will instantiate the procedure variables $\bar{g}$ to the actual procedures $\bar{q}$. For the environment variables $\bar{u}$, we will instantiate u_k to a new environment variable r_k', where r_k' is constrained to be a code for the procedure q_k. The result of these substitutions will be

$$(\ (\ \bigwedge_{k=1}^{|\tau|} q_k \ll r_k')\ \rightarrow\ \mathrm{pc}[\ p_i(\bar{q}, \bar{x})\ ]\ ),$$

where pc is now a pca involving the call $p_i(\bar{q}, \bar{x})$ with the actual procedures $\bar{q}$.

Roughly speaking, we would like to show that the formula on the left of the arrow is provable from the hypothesis $P^S \wedge R^S$, i.e.

(1) $\vdash P^S \wedge R^S \rightarrow (\bigwedge_{k=1}^{|\tau|} q_k \ll r'_k)$.

From line (1) we would be able to deduce

$$\vdash P^S \wedge R^S \rightarrow pc[\, p_i(\bar{q}, \bar{x})\,],$$

which would then lead to the proof of the main lemma in this case. By assumption, all the q_k are either declared or formal procedure names whose most general formulas are included in $P^S \wedge R^S$. But, observe that (1) is not semantically true in I because the values of the free environment variables r' are not constrained by $P^S \wedge R^S$.

The most general formulas in $P^S \wedge R^S$ relate the procedures in $\bar{q}$ to the free environment variables $\bar{r}^*$. For instance, R_k is $r_k \ll r_k^*$, so if r_k appears in $\bar{q}$, it is sufficient to constrain the corresponding variable in $\bar{r}'$ to be equal to r_k^*. This will satisfy the conjunct of $\bar{q} \ll \bar{r}'$ involving r_k. For a declared procedure p_k which appears in $\bar{q}$, we have the hypothesis P_k, which relates p_k to the code variables $\bar{r}^*$. If p_k has only ground parameters, then P_k asserts that $p_k(\bar{x})$ is less defined than $(p_k(\bar{x}))^*$, where $(p_k(\bar{x}))^*$ depends on $\bar{r}^*$. If p_k has a higher type, P_k is a generalization of this assertion. So, if p_k appears in $\bar{q}$, we need to constrain the corresponding variable in $\bar{r}'$ to be a code for p_k, given the values of $\bar{r}^*$. In section 7b of the simulation we described how to compute a code for p_k from the code variables $\bar{r}^*$. We can find a first-order formula that expresses the relationship between $\bar{r}'$ and $\bar{r}^*$ by taking strongest postconditions of the computation described in 7a (for formal procedures) and 7b (for declared procedures).

We will define a first-order formula $C(\bar{r}', \bar{r}^*)$ which constrains the value of $\bar{r}'$ in the necessary way to satisfy formula (1). Then, by Axiom 13, we will be able to deduce

$$\vdash P^S \wedge R^S \rightarrow \Big(C \rightsquigarrow \Big((\bigwedge_{k=1}^{|\tau|} q_k \ll r'_k) \rightarrow pc[\, p_i(\bar{q}, \bar{x})\,] \Big) \Big).$$

By the definition of $C \rightsquigarrow H$, this is equivalent to

$$\vdash P^S \wedge R^S \rightarrow \Big((\bigwedge_{k=1}^{|\tau|} (C \rightsquigarrow q_k \ll r'_k)) \rightarrow (C \rightsquigarrow pc[\, p_i(\bar{q}, \bar{x})\,]) \Big).$$

But, with the constraint expressed by C, it will be possible to show

$$\vdash P^S \wedge R^S \rightarrow (\bigwedge_{k=1}^{|\tau|} (C \rightsquigarrow q_k \ll r'_k)),$$

which is weaker than (1). Then, we will be able to deduce

$$\vdash P^S \wedge R^S \rightarrow (C \rightsquigarrow pc[\, p_i(\bar{q}, \bar{x})\,]).$$

Finally, we will complete this case of the proof of the main lemma by proving in the axiom system that $C \rightsquigarrow pc[\ p_i(\bar{q}, \bar{x})\ ]$ implies that $p_i(\bar{q}, \bar{x})$ is less defined than $(p_i(\bar{q}, \bar{x}))^*$. This is essentially what we need to prove in this case of the main lemma.

We now provide the formal details of this argument. The first step is to use Axiom 11 to instantiate the bound variables in P_i, with the substitutions $[r'_1/u_1,...,r'_{|\tau|}/u_{|\tau|}]$ and $[\bar{q}/\bar{g}]$. This gives us

$$(2)\quad \vdash P_i \rightarrow ((\bigwedge_{k=1}^{|\tau|} q_k \ll r'_k) \rightarrow rhs'),$$

where rhs′ is the pca on the right side in the definition of P_i with substitutions for $\bar{u}$ and $\bar{g}$. Applying the substitutions gives

$$(3)\quad rhs' \overset{def}{=}$$
$$\{\bar{x} = \bar{x}0 \wedge W(\bar{r}') \wedge W(\bar{r}^{*g})\}\ p_i(\bar{q}, \bar{x})$$
$$\{\exists b\ SP\ [p_i^*(\bar{r}', \bar{r}^{*g}; \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{r}') \wedge W(\bar{r}^{*g})]\}.$$

When we defined the simulation of procedure calls, we introduced the new local variables $\bar{r}'$, which hold the codes for the actual parameters of the call. We defined a program that assigns values $\bar{r}'$, using the values of the variables $\bar{r}^*$ as inputs. This program is called $\bar{r}' := e_S(\bar{r}^*)$.

Let $C(\bar{r}', \bar{r}^*)$ be a first-order formula that expresses $\exists b\ SP\ [\bar{r}' := e_S(\bar{r}^*);\ W(\bar{r}^*)]$. All of the free variables in C are environment variables. This allows C to be used as the first order formula in Axiom 13. We will use the properties of C from Lemma 9.

The next step is to apply Axiom 13 with the first-order formula C and the formula $((\bigwedge_{k=1}^{|\tau|} q_k \ll r'_k) \rightarrow rhs')$. Since $\vdash P^S \rightarrow P_i$, we can infer from (2)

$$\vdash P^S \wedge R^S \rightarrow (C \rightsquigarrow ((\bigwedge_{k=1}^{|\tau|} q_k \ll r'_k) \rightarrow rhs')).$$

If p_i has procedure parameters, so that $|\tau| \neq 0$, then by the definition of $C \rightsquigarrow H$, the above formula is equivalent to

$$\vdash P^S \wedge R^S \rightarrow \left((\bigwedge_{k=1}^{|\tau|} (C \rightsquigarrow q_k \ll r'_k)) \rightarrow (C \rightsquigarrow rhs') \right).$$

The next step is to show $\vdash P^S \wedge R^S \rightarrow (C \rightsquigarrow q_k \ll r'_k)$, for $k = 1,...,|\tau|$. Afterwards, we will be able to deduce

$$\vdash P^S \wedge R^S \rightarrow (C \rightsquigarrow rhs').$$

Lemma 10. Let $S = q_0(\bar{q}, \bar{z})$ be an arbitrary procedure of type $\tau = (\tau_1,...,\tau_{|\tau|}, \text{var}^{\|\tau\|})$. Let C be a first-order formula that expresses $\exists b\ SP[\bar{r}' := e(\bar{r}^*);\ W(\bar{r}^*)]$, where $\bar{r}' := e_S(\bar{r}^*)$ is the program that computes the codes $\bar{r}'$ in the simulation $(q_0(\bar{q}, \bar{z}))^*$. Then $\vdash P^S \wedge R^S \rightarrow (C \rightsquigarrow q_k \ll r'_k)$, for $k = 1,...,|\tau|$.

Proof. The lemma is vacuously true if $|\tau| = 0$. Otherwise, we must consider two cases depending on whether q_k is a formal or a declared procedure.

Case 1. Suppose q_k is a formal, r_h, for some h. Since this means r_h is free in the call $S = q_0(\bar{q}, \bar{z})$, the formula R_h is a conjunct of R^S. Since q_0 has type $\tau = (\tau_1,...,\tau_{|\tau|}, \text{var}^{\|\tau\|})$, r_h must have type τ_k. Thus by definition, R_h is $r_h \ll_{\tau_k} r_h^*$.

We would like to show $\vdash R_h \rightarrow (C \rightsquigarrow q_k \ll r'_k)$. In this case, since q_k is actually r_h, the formula to be proven is

$$\vdash r_h \ll r_h^* \rightarrow (C \rightsquigarrow r_h \ll r'_k).$$

By Axiom 13,

$$\vdash r_h \ll r_h^* \rightarrow (C \rightsquigarrow r_h \ll r_h^*).$$

Notation. For readability, we let quantifiers range over the smallest formula following them. For instance, $\exists x\ P \wedge Q$ should be read as $(\exists x\ P) \wedge Q$.

We will now show that $\vdash (C \rightsquigarrow r_h \ll r_h^*) \rightarrow (C \rightsquigarrow r_h \ll r'_k)$.

The right side of $(C \rightsquigarrow r_h \ll r_h^*)$ is the pca,

$$\{\bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r_h^*) \wedge C\}\ r_h(\bar{g}, \bar{x})$$
$$\{\exists b\ SP[<r_h^*>(<\bar{u}>, \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r_h^*)] \wedge C\}.$$

On the right side of $(C \rightsquigarrow r_h \ll r'_k)$ is the pca,

$$\{\bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r'_k) \wedge C\}\ r_h(\bar{g}, \bar{x})$$
$$\{\exists b\ SP[<r'_k>(<\bar{u}>, \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r'_k)] \wedge C\},$$

which is the same, except for occurrences of r_h^* and r'_k.

We will now consider the first-order formulas in these pcas, in order to use rule of consequence. We will use the fact from Lemma 9.3, $I \models C \supset r'_k = r_h^*$. On the left sides of the pcas, this implies that

$$I \models (\bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r_h^*) \wedge C) \equiv (\bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r'_k) \wedge C).$$

On the right sides, since C implies $r_h^* = r'_k$, it is obvious that

$$I \models C \supset (\exists b\ SP[<r_h^*>(<\bar{u}>, \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r_h^*)] \equiv$$
$$\exists b\ SP[<r_k'>(<\bar{u}>, \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r_k')]).$$

Rearranging the logical connectives gives

$$I \models (\exists b\ SP[<r_h^*>(<\bar{u}>, \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r_h^*)] \wedge C) \equiv$$
$$(\exists b\ SP[<r_k'>(<\bar{u}>, \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r_k')] \wedge C).$$

By the rule of consequence for pcas, Axiom 7, we can infer that the following formula is provable:

(4) $\vdash$
$\{\bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r_h^*) \wedge C\}\ r_h(\bar{g}, \bar{x})$
$\{\exists b\ SP[<r_h^*>(<\bar{u}>, \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r_h^*)] \wedge C\}$
$\rightarrow$
$\{\bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r_k') \wedge C\}\ r_h(\bar{g}, \bar{x})$
$\{\exists b\ SP[<r_k'>(<\bar{u}>, \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r_k')] \wedge C\}.$

Using first-order reasoning, one can easily show that

(D) $\vdash (A \rightarrow B)$ implies $\vdash (\forall \bar{g}\ \forall \bar{u}(H \rightarrow A)) \rightarrow (\forall \bar{g}\ \forall \bar{u}(H \rightarrow B))$,

where A, B, and H are arbitrary formulas, $\bar{g}$ is a list of procedure variables and $\bar{u}$ is a list of environment variables. We will call this derived rule (D).

We will now use rule (D) to show that $\vdash (C \rightsquigarrow r_h \ll r_h^*) \rightarrow (C \rightsquigarrow r_h \ll r_k')$. Formula A in the rule will be the lhs of formula (4). Note that this is also the pca on the rhs of $(C \rightsquigarrow r_h \ll r_h^*)$. Formula B in the rule will be the rhs of formula (4); this is the rhs of $(C \rightsquigarrow r_h \ll r_k')$. Thus, we have already shown $\vdash A \rightarrow B$.

For the formula H in the rule, we will use the formula on the left side of the main arrow in $(C \rightsquigarrow r_h \ll r_h^*)$. Notice that by the definition of $q \ll v$, if q0 and q1 are procedure identifiers of the same type, then the two formulas $(q0 \ll v0)$ and $(q1 \ll v1)$ have the same formula on the left side of the main arrow. This formula contains only variables that are universally quantified in the formula. We will use this formula as H in rule (D).

Using rule (D) and the provability of (4), we obtain

$\vdash (C \rightsquigarrow r_h \ll r_h^*) \rightarrow (C \rightsquigarrow r_h \ll r_k')$.

Since $\vdash r_h \ll r_h^* \rightarrow (C \rightsquigarrow r_h \ll r_h^*)$, we can conclude

$\vdash R_h \rightarrow (C \rightsquigarrow r_h \ll r_k')$,

as required in this case. This completes the case that q_k is a formal procedure.

Case 2. Now we have to make a similar argument for the case that q_k is a declared procedure, say p_h, for some h. Since p_h is free in the call $S = q_0(\bar{q}, \bar{z})$, P_h is a conjunct of P^S. Since we assumed that q_0 has type τ, p_h must have type τ_k.

We must show $\vdash P_h \rightarrow (C \rightsquigarrow p_h \ll r'_k)$.

The first-order formula C will be used again. As before, we start by using Axiom 13 to get

$\vdash P_h \rightarrow (C \rightsquigarrow P_h)$.

As before, we will use rule (D). Since p_h has type τ_k, by the definition of P_h, the lhs of $(C \rightsquigarrow P_h)$ is the same as the lhs of $(C \rightsquigarrow p_h \ll r'_k)$.

In order to use the rule we must show

$\vdash$ rhs $(C \rightsquigarrow P_h) \rightarrow$ rhs $(C \rightsquigarrow p_h \ll r'_k)$.

It will then follow from rule (D) that $\vdash P_h \rightarrow (C \rightsquigarrow p_h \ll r'_k)$.

By definition, rhs $(C \rightsquigarrow p_h \ll r'_k)$ is the pca

$\{\bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r'_k) \wedge C\}$
$p_h(\bar{g}, \bar{x})$
$\{\exists b\ SP\,[<r'_k>(<\bar{u}>, \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r'_k)] \wedge C\}$.

On the other hand, rhs $(C \rightsquigarrow P_h)$ is the pca

$\{\bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(\bar{r}^{*g}) \wedge C\}$
$p_h(\bar{g}, \bar{x})$
$\{\exists b\ SP\,[p^*_h(\bar{u}, \bar{r}^{*g}; \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(\bar{r}^{*g})] \wedge C\}$.

As before, since both pcas have the same statement in the middle, we will use rule of consequence. First, consider the formulas on the left sides of the two pcas. Here, we will make use of the fact that $I \models C \supset W(\bar{r}^*) \wedge W(\bar{r}')$, proved in Lemma 9.1. From this, it follows that the first-order formulas on the left sides of the two pcas are equivalent.

Now we will show that the first-order formulas on the right sides of the pcas are also equivalent, using Lemma 9.4. By the definition of the first-order formula C, C in interpretation I implies that r'_k is a code for the declaration of p_h in an environment containing the other declared procedures reachable by following free procedures and containing declarations of the formal procedures encoded by the variables $\bar{r}^{*g}$ p_h. Intuitively, since the procedure p^*_h is defined to simulate calls on p_h and since C implies

$W(\bar{r}')$ and $W(\bar{r}^*)$, one can see that

$$I \models C \supset (\exists b\ SP\,[<r'_k>(<\bar{u}>, \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r'_k)] \equiv$$
$$\exists b\ SP\,[p^*_h(\bar{u}, \bar{r}^{*g}; \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(\bar{r}^{*g})])$$

(from Lemma 9.4.).

As in the previous case, rearranging C gives

$$I \models (\exists b\ SP\,[<r'_k>(<\bar{u}>, \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(r'_k)] \wedge C) \equiv$$
$$(\exists b\ SP\,[p^*_h(\bar{u}, \bar{r}^{*g}; \bar{x});\ \bar{x} = \bar{x}0 \wedge W(\bar{u}) \wedge W(\bar{r}^{*g})] \wedge C).$$

Now by rule of consequence for pcas, we can deduce $\vdash$ rhs $(C \rightsquigarrow p_h \ll r'_k) \rightarrow$ rhs $(C \rightsquigarrow p_h \ll r'_k)$.

Finally, using rule (D), we can deduce $\vdash (C \rightsquigarrow P_h) \rightarrow (C \rightsquigarrow p_h \ll r'_k)$, as required in this case. This completes the case that q_k is a declared procedure. □

We have now completed the proof that $\vdash P^S \wedge R^S \rightarrow (C \rightsquigarrow q_k \ll r'_k)$, for $k = 1,...,|\tau|$. As we outlined in the beginning, this allows us to deduce that $\vdash P^S \wedge R^S \rightarrow (C \rightsquigarrow$ rhs′). Recall that rhs′ is the pca on line (3). Expanding the definition of $(C \rightsquigarrow$ rhs′), we have shown that

$$\vdash P^S \wedge R^S \rightarrow$$
$$\{\bar{x}{=}\bar{x}0 \wedge W(\bar{r}') \wedge W(\bar{r}^{*g}) \wedge C\}$$
$$p_i(\bar{q}, \bar{x})$$
$$\{\exists b\ SP\,[p^*_i(\bar{r}', \bar{r}^{*g}; \bar{x});\ \bar{x}{=}\bar{x}0 \wedge W(\bar{r}') \wedge W(\bar{r}^{*g})] \wedge C\}.$$

Next, we use rule of consequence to simplify the pre- and postconditions in the pca in the above formula. From Lemma 9.1, we get

$$I \models (\bar{x}{=}\bar{x}0 \wedge W(\bar{r}') \wedge W(\bar{r}^{*g}) \wedge C) \equiv (\bar{x}{=}\bar{x}0 \wedge C).$$

To simplify the postcondition, observe that the following equivalences hold:

$I \models \exists b\ SP\,[p_i^*(\bar{r}', \bar{r}^{*g}; \bar{x});\ \bar{x}=\bar{x}0 \wedge W(\bar{r}') \wedge W(\bar{r}^{*g})] \wedge C$

$\equiv \exists b\ (SP\,[p_i^*(\bar{r}', \bar{r}^{*g}; \bar{x});\ \bar{x}=\bar{x}0 \wedge W(\bar{r}') \wedge W(\bar{r}^{*g})] \wedge C)$
(because b is not free in C)

$\equiv \exists b\ SP\,[p_i^*(\bar{r}', \bar{r}^{*g}; \bar{x});\ \bar{x}=\bar{x}0 \wedge W(\bar{r}') \wedge W(\bar{r}^{*g}) \wedge C]$
(by property SP 1 of strongest postconditions)

$\equiv \exists b\ SP\,[p_i^*(\bar{r}', \bar{r}^{*g}; \bar{x});\ \bar{x}=\bar{x}0 \wedge C]$.
(by Lemma 9.1)

Thus, by the rule of consequence,

$\vdash P^S \wedge R^S \rightarrow \{\bar{x}=\bar{x}0 \wedge C\}\ p_i(\bar{q}, \bar{x})\ \{\exists b\ SP\,[p_i^*(\bar{r}', \bar{r}^{*g}; \bar{x});\ \bar{x}=\bar{x}0 \wedge C]\}.$

Now, in order to introduce the actual ground variables of the call $p_i(\bar{q}, \bar{z})$, we apply the substitution $[\bar{z}/\bar{x}, \bar{z}0/\bar{x}0]$ in the above formula by Ax 12. Since the variables $\bar{x}$ and $\bar{x}0$ do not appear free in C, C is left unchanged by the substitution. The result is

$\vdash P^S \wedge R^S \rightarrow \{\bar{z}=\bar{z}0 \wedge C\}\ p_i(\bar{q}, \bar{z})\ \{\exists b\ SP\,[p_i^*(\bar{r}', \bar{r}^{*g}; \bar{z});\ \bar{z}=\bar{z}0 \wedge C]\}.$

The next step is to use Rule R3 to existentially quantify over the environment variables $\bar{r}'$ on both sides of the pca. This gives us

$\vdash P^S \wedge R^S \rightarrow \{\exists \bar{r}'\ (\bar{z}=\bar{z}0 \wedge C)\}\ p_i(\bar{q}, \bar{z})\ \{\exists \bar{r}'\ (\exists b\ SP\,[p_i^*(\bar{r}', \bar{r}^{*g}; \bar{z});\ \bar{z}=\bar{z}0 \wedge C])\}$

For the precondition, we have

$I \models \exists \bar{r}'\ (\bar{z}=\bar{z}0 \wedge C)$

$\equiv \bar{z}=\bar{z}0 \wedge \exists \bar{r}'\ C$

$\equiv \bar{z}=\bar{z}0 \wedge W(\bar{r}^*).$
(by Lemma 9.2)

So by rule of consequence,

(5) $\vdash P^S \wedge R^S \rightarrow$
$\{\bar{z}=\bar{z}0 \wedge W(\bar{r}^*)\}\ p_i(\bar{q}, \bar{z})\ \{\exists \bar{r}'\ (\exists b\ SP\,[p_i^*(\bar{r}', \bar{r}^{*g}; \bar{z});\ \bar{z}=\bar{z}0 \wedge C])\}.$

The Main Lemma to be proven is

(6) $\vdash P^S \wedge R^S \rightarrow \{\bar{z}=\bar{z}0 \wedge W(\bar{r}^*)\}\ p_i(\bar{q}, \bar{z})\ \{\exists b\ SP\,[(p_i(\bar{q}, \bar{z}))^*;\ \bar{z}=\bar{z}0 \wedge W(\bar{r}^*)]\}.$

The remaining step in getting from the pca in (5) to the pca in (6) is to relate the

postconditions of the two pcas. Observe that the postcondition in (6) involves the strongest postcondition of the simulation $(p_i(\bar{q}, \bar{z}))^*$, while the postcondition in (5) involves a call on the procedure p_i^*, which is part of the simulation. The formula C expresses the strongest postcondition of the program that computes the code values $\bar{r}'$ in the simulation. We will now prove a technical lemma which states that the two postconditions are actually equivalent.

Lemma 11. (i) Consider a procedure call $S = p_i(\bar{q}, \bar{z})$. Let $C \equiv \exists b\ SP[\bar{r}' := e_S(\bar{r}^*);\ W(\bar{r}^*)]$, where $\bar{r}' := e_S(\bar{r}^*)$ is the program that computes the codes of the procedures passed as parameters in the simulation $(p_i(\bar{q}, \bar{z}))^*$. Then

$$I \models \exists b\ SP[(p_i(\bar{q}, \bar{z}))^*;\ \bar{z}=\bar{z}0 \wedge W(\bar{r}^*)] \equiv \exists \bar{r}'\ (\exists b\ SP[p_i^*(\bar{r}', \bar{r}^{*g};\ \bar{z});\ \bar{z}=\bar{z}0 \wedge C]).$$

(ii) For a call on a formal procedure, $r_i(\bar{q}, \bar{z})$, the analogous property holds,

$$I \models \exists b\ SP[(r_i(\bar{q}, \bar{z}))^*;\ \bar{z}=\bar{z}0 \wedge W(\bar{r}^*)] \equiv \exists \bar{r}'\ (\exists b\ SP[<r_i^*>(<\bar{r}'>, \bar{z});\ \bar{z}=\bar{z}0 \wedge C]).$$

Proof. We will consider the case (i) of a call on a declared procedure; the other case follows by exactly the same argument.

First observe that the following equivalences hold:

(7) $I \models \exists b\ SP[p_i^*(\bar{r}', \bar{r}^{*g};\ \bar{z});\ \bar{z}=\bar{z}0 \wedge C]$

$\equiv \exists b\ SP[p_i^*(\bar{r}', \bar{r}^{*g};\ \bar{z});\ \bar{z}=\bar{z}0 \wedge \exists b\ SP[\bar{r}' := e_S(\bar{r}^*);\ W(\bar{r}^*)]]$
(definition of C)

$\equiv \exists b\ SP[p_i^*(\bar{r}', \bar{r}^{*g};\ \bar{z});\ \exists b\ SP[\bar{r}' := e_S(\bar{r}^*);\ \bar{z}=\bar{z}0 \wedge W(\bar{r}^*)]]$
(by property SP 1 of strongest postconditions)

(8) $\equiv \exists b\ SP[(\bar{r}' := e_S(\bar{r}^*);\ p_i^*(\bar{r}', \bar{r}^{*g};\ \bar{z}));\ \bar{z}=\bar{z}0 \wedge W(\bar{r}^*)]$.
(by Lemma 8)

Now we will use property SP 5 of strongest postconditions, which states that $SP[(\text{Begin var x; S End});\ P] \equiv \exists x(SP[S;\ P])$, provided the variable x is not free in the formula P. From the deduction that line (7) $\equiv$ line (8) and from the definition of $(p_i(\bar{q}, \bar{z}))^*$, it follows that

$I \models \exists \bar{r}' \, (\exists b \; SP\,[p_i^*(\bar{r}', \bar{r}^{*g}; \bar{z});\;\; \bar{z}{=}\bar{z}0 \wedge C]\,)$
(this is $\exists \bar{r}'$ (7))

$\equiv \exists \bar{r}' \, (\exists b \; SP\,[(\bar{r}' := e_S(\bar{r}^*);\; p_i^*(\bar{r}', \bar{r}^{*g}; \bar{z}));\;\; \bar{z}{=}\bar{z}0 \wedge W(\bar{r}^*)]\,)$
(this is $\exists \bar{r}'$ (8))

$\equiv \exists b \; SP\,[(\text{Begin var } \bar{r}';\; \bar{r}' := e_S(\bar{r}^*);\; p_i^*(\bar{r}', \bar{r}^{*g}; \bar{z})\; \text{End});\;\; \bar{z}{=}\bar{z}0 \wedge W(\bar{r}^*)]$
(by property SP 5)

$\equiv \exists b \; SP\,[(p_i(\bar{q}, \bar{z}))^*;\;\; \bar{x}{=}\bar{x}0 \wedge W(\bar{r}^*)]\,.$
(definition of $(p_i(\bar{q}, \bar{z}))^*$)

This completes the proof of Lemma 11. □

Using the first-order equivalence from the lemma with the rule of consequence, we can infer

$$\vdash P^S \wedge R^S \rightarrow \{\bar{z}{=}\bar{z}0 \wedge W(\bar{r}^*)\}\; p_i(\bar{q}, \bar{z})\; \{\exists b \; SP\,[(p_i(\bar{q}, \bar{z}))^*;\;\; \bar{z}{=}\bar{z}0 \wedge W(\bar{r}^*)]\}.$$

This completes the proof of the Main Lemma for the case of a call on a declared procedure. □

7.3. Proof of the Main Lemma for a call on a formal procedure

Let $S = r_i(\bar{q}, \bar{z})$, where r_i is a formal procedure of type τ, $\tau = (\tau_1,...,\tau_{|\tau|}, \text{var}^{\|\tau\|})$. This case is similar to a call on a declared procedure.

By assumption, the formula R_i is a conjunct of R^S. As before, we begin by using first-order reasoning to instantiate the bound variables in a general formula to the specifics of this call. Recall that R_i is $r_i \ll r_i^*$. We instantiate the bound variables in R_i with the substitutions $[r'_1/u_1,...,r'_{|\tau|}/u_{|\tau|}]$, and $[\bar{q}^{|\tau|}/\bar{g}^{|\tau|}]$, giving

$$\vdash R_i \rightarrow$$
$$((\bigwedge_{k=1}^{|\tau|} q_k \ll r'_k) \rightarrow$$
$$\{\bar{x}=\bar{x}0 \wedge W(\bar{r}') \wedge W(r_i^*)\}\ r_i(\bar{q}, \bar{x})$$
$$\{\exists b\ SP[\ <r_i^*>(<\bar{r}'>, \bar{x});\ \bar{x}=\bar{x}0 \wedge W(\bar{r}') \wedge W(r_i^*)]\}).$$

Let C be a first-order formula that expresses $\exists b\ SP[\bar{r}' := e_S(\bar{r}^*);\ W(\bar{r}^*)]$ for the program $\bar{r}' := e_S(\bar{r}^*)$ in the simulation $(r_i(\bar{q}, \bar{z}))^*$. Now, using the derived $\rightsquigarrow$ axiom with the first-order formula C gives us

$\vdash R_i \rightarrow ((A_1 \wedge ... \wedge A_{|\tau|}) \rightarrow B)$, where A_k is $C \rightsquigarrow (q_k \ll r'_k)$, for $k=1,...,|\tau|$, and B is

$$\{\bar{x}=\bar{x}0 \wedge W(\bar{r}') \wedge W(r_i^*) \wedge C\}\ r_i(\bar{q}, \bar{x})$$
$$\{\exists b\ SP[\ <r_i^*>(<\bar{r}'>, \bar{x});\ \bar{x}=\bar{x}0 \wedge W(\bar{r}') \wedge W(r_i^*)] \wedge C\}.$$

Note that the formulas C and the A_k are the same as in the case of a call on a declared procedure. The formula B is different since it contains the strongest postcondition of $<r_i^*>(<\bar{r}'>, \bar{x})$, a simulated call using an interpreter program, while in the case of a call on a declared procedure, we had a pca with the strongest postcondition of a call on p_i^*. This reflects the difference in the simulation of the two kinds of calls.

As before, the next step is to use Lemma 10 to deduce

$$\vdash P^S \wedge R^S \rightarrow (A_1 \wedge ... \wedge A_{|\tau|}).$$

Therefore we have

$$\vdash P^S \wedge R^S \rightarrow B.$$

Next we use Rule R3 to existentially quantify over $\bar{r}'$ on both sides of B. The result is

$$\vdash P^S \wedge R^S \rightarrow$$
$$\{\exists \bar{r}' \, (\bar{x}=\bar{x}0 \wedge W(\bar{r}') \wedge W(r_i^*) \wedge C)\}$$
$$r_i(\bar{q}, \bar{x})$$
$$\{\exists \bar{r}' \, (\exists b \; SP [\langle r_i^* \rangle(\langle \bar{r}' \rangle, \bar{x}); \; \bar{x}=\bar{x}0 \wedge W(\bar{r}') \wedge W(r_i^*)] \wedge C)\}.$$

On the left side of the pca, observe that

$$I \models \exists \bar{r}' \, (\bar{x}=\bar{x}0 \wedge W(\bar{r}') \wedge W(r_i^*) \wedge C) \equiv \bar{x}=\bar{x}0 \wedge W(\bar{r}^*),$$

by Lemma 9.1 and 9.2.

For the right side, first observe that

$$I \models \exists \bar{r}' \, (\exists b \; SP [\langle r_i^* \rangle(\langle \bar{r}' \rangle, \bar{x}); \; \bar{x}=\bar{x}0 \wedge W(\bar{r}') \wedge W(r_i^*)] \wedge C)$$
$$\equiv \exists \bar{r}' \, (\exists b \; SP [\langle r_i^* \rangle(\langle \bar{r}' \rangle, \bar{x}); \; \bar{x}=\bar{x}0 \wedge C]),$$

because we can move C inside the $\exists b$ and the strongest postcondition.

We showed in Lemma 11 that

$$I \models \exists \bar{r}' \, (\exists b \; SP [\langle r_i^* \rangle(\langle \bar{r}' \rangle, \bar{x}); \; \bar{x}=\bar{x}0 \wedge C]) \equiv \exists b \; SP [(r_i(\bar{q}, \bar{x}))^*; \; \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)].$$

By rule of consequence, we can now deduce

$$\vdash P^S \wedge R^S \rightarrow \{\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)\} \; r_i(\bar{q}, \bar{x}) \; \{\exists b \; SP [(r_i(\bar{q}, \bar{x}))^*; \; \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}.$$

Finally, Axiom 11 can be used with the substitution $[\bar{z}/\bar{x}]$ to rename the ground variables.

This completes the proof of the Main Lemma in the case of a call on a formal procedure. □

7.4. Proof of the Main Lemma for procedure declarations

Suppose S = E|S1, where E is an environment and S1 is a statement. Assume that E has the form $\{p_1(\text{formal-list}_1) \leftarrow \text{body}_1, \ldots, p_n(\text{formal-list}_n) \leftarrow \text{body}_n\}$.

By definition,

$P^{S1} = P^S \wedge (P_1 \wedge \ldots \wedge P_n)$,
$R^{S1} = R^S$,
$P^{body_i} = P^S \wedge (P_1 \wedge \ldots \wedge P_n)$, for i=1,...,n,
$R^{body_i} = R^S \wedge R^{f\,p_i}$, for i=1,...,n,
(where $R^{f\,p_i} = (\ \bigwedge_j R_j \mid r_j$ is in formal-list of $p_i)$).

By the inductive hypothesis, we can assume

$\vdash P^S \wedge (P_1 \wedge \ldots \wedge P_n) \wedge R^S \wedge R^{f\,p_i} \rightarrow$
$\{\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)\}\ body_i\ \{\exists b\ SP[body_i^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}$, for i=1,...,n, and

$\vdash P^S \wedge (P_1 \wedge \ldots \wedge P_n) \wedge R^S \rightarrow$
$\{\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)\}\ S1\ \{\exists b\ SP[S1^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}$.

The main idea in this case is to use the Recursion Rule R3 to prove that the formulas $P_1,\ldots,P_n$ hold in the environment E. Then it is straightforward to show

$\vdash P^S \wedge R^S \rightarrow \{\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)\}\ S\ \{\exists b\ SP[S^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}$.

First of all, by the definition of p_i^*, $SP[body_i^*;\ Q] \equiv SP[p_i^*(\bar{r}^{*\,all\,p_i}, \bar{x});\ Q]$, for any first order-formula Q. Using this fact, we can rewrite the inductive hypotheses for the $body_i$ to

(9) $\vdash P^S \wedge (P_1 \wedge \ldots \wedge P_n) \wedge R^S \wedge R^{f\,p_i} \rightarrow$
$\{\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)\}\ body_i\ \{\exists b\ SP[p_i^*(\bar{r}^{*\,all\,p_i}, \bar{x});\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}$, for i=1,...,n.

The next step is to existentially quantify over $\bar{r}^*$ variables that are not used by p_i^*. We define $\bar{r}^{*\,comp\,p_i}$ (for complement p_i) to be the sequence of variables in $\bar{r}^*$ that are not in $\bar{r}^{*\,all\,p_i}$. We will apply rule R2 to line (9). In order to do this, we must check that none of the variables in $\bar{r}^{*\,comp\,p_i}$ are free in the formula to the left of the arrow. First consider the free environment variables of $P^S \wedge (P_1 \wedge \ldots \wedge P_n)$. By definition, the formula P_i has free occurrences of the variables $\bar{r}^{*\,g\,p_i}$. The formula P^S contains assertions about the declared procedures that have S in their scope. Clearly, any

variable in $\bar{r}^{*g}$ for one of these procedures must also be in $\bar{r}^{*g\ p_i}$, and so it is not in the complement. Similarly, none of the environment variables free in $(P_1 \wedge \ldots \wedge P_n)$ can be in the complement. It remains to consider $R^S \wedge R^{f\ p_i}$. By definition, R_i is a formula whose only free environment variable is r_i^*. The formula R^S contains assertions for all of the procedures r_i that have S in their scope. Clearly, all of the environment variables free in R^S must be in $\bar{r}^{*g\ p_i}$ and not in the complement. Finally, the environment variables free in $R^{f\ p_i}$ are exactly $\bar{r}^{*f\ p_i}$ and are not in the complement. Thus we can apply rule R2 to deduce

$$\vdash P^S \wedge (P_1 \wedge \ldots \wedge P_n) \wedge R^S \wedge R^{f\ p_i} \rightarrow$$
$$\{\exists \bar{r}^{*comp\ p_i}(\bar{x}=\bar{x}0 \wedge W(\bar{r}^*))\}\ body_i$$
$$\{\exists \bar{r}^{*comp\ p_i}(\exists b\ SP\,[p_i^*(\bar{r}^{*all\ p_i}, \bar{x});\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)])\},$$

for i=1,...,n.

Now, observe that because $\exists \bar{x}\ W(\bar{x})$ is true in I

$$I \models \exists \bar{r}^{*comp\ p_i}(\bar{x}=\bar{x}0 \wedge W(\bar{r}^*))$$
$$\equiv \exists \bar{r}^{*comp\ p_i}(\bar{x}=\bar{x}0 \wedge W(\bar{r}^{*all\ p_i}) \wedge W(\bar{r}^{*comp\ p_i}))$$
$$\equiv \bar{x}=\bar{x}0 \wedge W(\bar{r}^{*all\ p_i}) \wedge \exists \bar{r}^{*comp\ p_i}(W(\bar{r}^{*comp\ p_i}))$$
$$\equiv \bar{x}=\bar{x}0 \wedge W(\bar{r}^{*all\ p_i}), \quad \text{and}$$

$$I \models \exists \bar{r}^{*comp\ p_i}(\exists b\ SP\,[p_i^*(\bar{r}^{*all\ p_i}, \bar{x});\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)])$$
$$\equiv \exists b\ SP\,[p_i^*(\bar{r}^{*all\ p_i}, \bar{x});\ \exists \bar{r}^{*comp\ p_i}(\bar{x}=\bar{x}0 \wedge W(\bar{r}^*))]$$

(by property SP 3 of strongest postconditions)

$$\equiv \exists b\ SP\,[p_i^*(\bar{r}^{*all\ p_i}, \bar{x});\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^{*all\ p_i})].$$

So by the rule of consequence,

$$\vdash P^S \wedge (P_1 \wedge \ldots \wedge P_n) \wedge R^S \wedge R^{f\ p_i} \rightarrow$$
$$\{\bar{x}=\bar{x}0 \wedge W(\bar{r}^{*all\ p_i})\}\ body_i\ \{\exists b\ SP\,[p_i^*(\bar{r}^{*all\ p_i}, \bar{x});\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^{*all\ p_i})]\},$$

for i=1,...,n.

By first-order reasoning, this can be rearranged to get

$$\vdash P^S \wedge R^S \rightarrow ((P_1 \wedge \ldots \wedge P_n) \rightarrow$$
$$(\bigwedge_{i=1}^{n} (R^{f\,p_i} \rightarrow \{\bar{x}=\bar{x}0 \wedge W(\bar{r}^{*all\,p_i})\}$$
$$body_i$$
$$\{\exists b\ SP\,[p_i^*(\bar{r}^{*all\,p_i}, \bar{x});\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^{*all\,p_i})]\}))).$$

For the next step, we want to use first-order reasoning to universally quantify over $\bar{r}^{f\,p_i}$ and $\bar{r}^{*f\,p_i}$ to infer

$$(10)\ \vdash P^S \wedge R^S \rightarrow ((P_1 \wedge \ldots \wedge P_n) \rightarrow$$
$$(\bigwedge_{i=1}^{n} \forall\bar{r}^{f\,p_i}\,\forall\bar{r}^{*f\,p_i}\,(R^{f\,p_i} \rightarrow$$
$$\{\bar{x}=\bar{x}0 \wedge W(\bar{r}^{*all\,p_i})\}\ body_i$$
$$\{\exists b\ SP\,[p_i^*(\bar{r}^{*all\,p_i}, \bar{x});\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^{*all\,p_i})]\}))).$$

This will put the formula close to the form of the hypothesis of the recursion rule. By first-order reasoning, $\vdash H1 \rightarrow H2$ implies $\vdash H1 \rightarrow \forall r\ H2$, provided r is not free in H1. So, we must show that none of the variables in $\bar{r}^{f\,p_i}$ and $\bar{r}^{*f\,p_i}$ are free in $P^S \wedge (P_1 \wedge \ldots \wedge P_n) \wedge R^S$. None of the $\bar{r}^{f\,p_i}$ appear free in these formulas; for R^S, the reason is that we assumed that no formal procedure name appears in more than one formal-list in π, so none of the procedure names in $\bar{r}^{f\,p_i}$ can be global to $S = E|S1$, and so none of the $\bar{r}^{f\,p_i}$ can appear in R^S. Also, no formal procedure names appear free in any of the P_i. For the environment variables $\bar{r}^{*f\,p_i}$, none can appear in R^S for the same reason that the formal procedure names cannot appear. None of the $\bar{r}^{*f\,p_i}$ can appear in $P^S \wedge (P_1 \wedge \ldots \wedge P_n)$ because for each P_i, the only free environment variables are the $\bar{r}^{*g\,p_i}$, and none of the formals of $p_1,\ldots,p_n$ can be global to $p_1,\ldots,p_n$ or declared procedures global to S. Thus, the above formula (10) is provable by first-order reasoning.

The next step is to rename the bound variables so that the same universal variables are used in both $P_1 \wedge \ldots \wedge P_n$ and in the conjunction of the n formulas for $body_1$, ..., $body_n$. This renaming puts the formula into the exact form of the hypothesis of the recursion rule.

Now we use the recursion rule. The conclusion of the recursion rule is

$$\vdash P^S \wedge R^S \rightarrow$$
$$(\bigwedge_{i=1}^{n} \forall\bar{r}^{f\,p_i}\,\forall\bar{u}\,(R^{f\,p_i} \rightarrow$$
$$\{\bar{x}=\bar{x}0 \wedge W(\bar{r}^{*all})\}\ E|p_i(\bar{r}^{f}, \bar{x})\ \{\exists b\ SP\,[p_i^*(\bar{r}^{*all}, \bar{x});\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^{*all})]\})).$$

Renaming bound variables, this gives

(11) $\vdash P^S \wedge R^S \rightarrow (E|P_1 \wedge \ldots \wedge E|P_n)$.

From the inductive hypothesis, we had

$$\vdash P^S \wedge (P_1 \wedge \ldots \wedge P_n) \wedge R^S \rightarrow \{\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)\}\ S1\ \{\exists b\ SP[S1^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}.$$

Using Rule R2, we can infer

(12) $\vdash E|(P^S \wedge (P_1 \wedge \ldots \wedge P_n) \wedge R^S) \rightarrow \{\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)\}\ E|S1\ \{\exists b\ SP[S1^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}.$

Similarly, from R2 we can deduce $\vdash P^S \rightarrow E|P^S$, $\vdash R^S \rightarrow E|R^S$, because no procedure name bound in E is free in P^S, R^S.

Finally, combining these results with lines (11) and (12), we can deduce

$$\vdash P^S \wedge R^S \rightarrow \{\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)\}\ E|S1\ \{\exists b\ SP[S1^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}.$$

This completes the proof of the Main Lemma in the case of a block with a procedure declaration. □

7.5. Proof of the Main Lemma in the case (S1; S2)

For statements other than procedure calls and declarations, the proof of the Lemma is comparatively straightforward and closely parallels the relative completeness proofs that have been given for conventional Hoare axiom systems. As an example, we will prove the Main Lemma for the case S = (S1; S2).

By hypothesis,

$\vdash P^{S1} \wedge R^{S1} \rightarrow \{\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)\}\ S1\ \{\exists b\ SP[S1^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\},$
$\vdash P^{S2} \wedge R^{S2} \rightarrow \{\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)\}\ S2\ \{\exists b\ SP[S2^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}.$

By definition,

$$P^S \equiv P^{S1} \wedge P^{S2},\ R^S \equiv R^{S1} \wedge R^{S2}.$$

We proceed by putting the hypothesis for S2 into a form so that we can use the rule of concatenation. Let $\bar{x}'$ be a vector of fresh variables of the same length as $\bar{x}$ and $\bar{x}0$. By Axiom 11, we can apply the substitution $[\bar{x}'/\bar{x}0]$ in the hypothesis for S2, giving

$$\vdash P^{S2} \wedge R^{S2} \rightarrow \{\bar{x}=\bar{x}' \wedge W(\bar{r}^*)\}\ S2\ \{\exists b\ SP[S2^*;\ \bar{x}=\bar{x}' \wedge W(\bar{r}^*)]\}.$$

Now let $G \equiv \exists b\ SP[S1^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)][\bar{x}'/\bar{x}]$. Since G is a first-order formula that has no free variables that are free in the statement S2, we can use Axiom 9 to add G to both sides of the last pca to get

$$\vdash P^{S2} \wedge R^{S2} \rightarrow \{\bar{x}=\bar{x}' \wedge W(\bar{r}^*) \wedge G\}\ S2\ \{\exists b\ SP[S2^*;\ \bar{x}=\bar{x}' \wedge W(\bar{r}^*)] \wedge G\}.$$

Next, we use Axiom 8 to existentially quantify over the variables $\bar{x}'$, which are not free in S2,

$$\vdash P^{S2} \wedge R^{S2} \rightarrow$$
$$\{\exists \bar{x}'\ (\bar{x}=\bar{x}' \wedge W(\bar{r}^*) \wedge G)\}\ S2\ \{\exists \bar{x}'\ (\exists b\ SP[S2^*;\ \bar{x}=\bar{x}' \wedge W(\bar{r}^*)] \wedge G)\}.$$

In the above, by the definition of G,

(13) $I \models \exists \bar{x}'\ (\bar{x}=\bar{x}' \wedge W(\bar{r}^*) \wedge G)$

$$\equiv \exists \bar{x}'\ (\bar{x}=\bar{x}' \wedge \exists b\ SP[S1^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)][\bar{x}'/\bar{x}])$$
(by property SP 1 of strongest postconditions)

$$\equiv \exists b\ SP[S1^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)].$$

Using Lemma 8, about the monotonic property of the simulation in b, one can see that the following equivalences hold in I:

$I \models \exists \bar{x}' \, (\exists b \; SP\,[S2^*;\ \bar{x}=\bar{x}' \wedge W(\bar{r}^*)] \wedge G)$

$\equiv \exists \bar{x}' \, (\exists b \; SP\,[S2^*;\ \bar{x}=\bar{x}' \wedge W(\bar{r}^*) \wedge G])$
(by property SP 1 of strongest postconditions)

$\equiv \exists b \; SP\,[S2^*;\ \exists \bar{x}' \, (\bar{x}=\bar{x}' \wedge W(\bar{r}^*) \wedge G)]$
(by property SP 3 of strongest postconditions)

$\equiv \exists b \; SP\,[S2^*;\ \exists b \; SP\,[S1^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]]$
(by line (13))

$\equiv \exists b \; SP\,[(S1^*; S2^*);\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]$
(by the corollary to Lemma 8)

$\equiv \exists b \; SP\,[(S1; S2)^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]$.
(by definition of (S1; S2)*)

Thus by the rule of consequence,

$\vdash P^{S2} \wedge R^{S2} \rightarrow$
$\{\exists b \; SP\,[S1^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}\ S2\ \{\exists b \; SP\,[(S1; S2)^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}$.

Finally, using the rule of concatenation with the above formula and the inductive hypothesis for S1, we can show

$\vdash P^{S2} \wedge R^{S2} \rightarrow \{\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)\}\ S1;S2\ \{\exists b \; SP\,[(S1; S2)^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}$.

This completes the proof of the Main Lemma in the case S = (S1; S2). □

7.6. Proof of the Completeness Theorem.

We can now complete the proof of the Theorem. From the Main Lemma, we have

$$\vdash P^{\pi} \wedge R^{\pi} \rightarrow \{\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)\}\ \pi\ \{\exists b\ SP[\pi^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}.$$

Recall that $P^{\pi} = R^{\pi} =$ True. Thus by arrow rules,

$$\vdash \{\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)\}\ \pi\ \{\exists b\ SP[\pi^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)]\}.$$

Now, we existentially quantify over $\bar{r}^*$ on both sides, using rule R3. On the left, we have the equivalence

$$I \models \exists \bar{r}^*\ (\bar{x}=\bar{x}0 \wedge W(\bar{r}^*)) \equiv \bar{x}=\bar{x}0.$$

On the right, observe that

$$I \models \exists \bar{r}^*\ (\exists b\ SP[\pi^*;\ \bar{x}=\bar{x}0 \wedge W(\bar{r}^*)])$$

$$\equiv \exists b\ SP[\pi^*;\ \exists \bar{r}^*(\bar{x}=\bar{x}0 \wedge W(\bar{r}^*))]$$
(by SP 3, because $\bar{r}^*$ is not free in π^*)

$$\equiv \exists b\ SP[\pi^*;\ \bar{x}=\bar{x}0]$$

$$\equiv SP[\pi;\ \bar{x}=\bar{x}0].$$
(by Lemma 7)

By the rule of consequence, we show

$$\vdash \{\bar{x}=\bar{x}0\}\ \pi\ \{SP[\pi;\ \bar{x}=\bar{x}0]\}.$$

Standard techniques [Go75,Cl79] can now be used to show $\vdash \{U\}\ \pi\ \{V\}$ for any first-order formulas U, V, such that $I \models \{U\}\ \pi\ \{V\}$.

Q.E.D!

8. Expressing Total Correctness

Once we have established that there is a sound and relatively complete axiom system for partial-correctness assertions for L4 programs, it is natural to ask whether there can be a "good" axiom system for deducing all the formulas of the full logic that are true in an interpretation, where the full logic contains partial correctness assertions, nesting of arbitrary formulas with $\rightarrow$, and quantification over environment variables. We answer this question in the negative by showing that it is possible to express total correctness in the full logic. Consider the formula

$$\forall \bar{v}\ (U(\bar{v}) \ \rightarrow \ (\{\bar{x}{=}\bar{v}\}\ \pi(\bar{x})\ \{\neg V(\bar{x})\} \ \rightarrow \ \text{False})),$$

where $\bar{v}$ is a list of environment variables, $\bar{x}$ is a list of program variables, and π is a program. In the above formula, the subformula $U(\bar{v})$ is an abbreviation for the pca $\{\text{True}\}\ x := x\ \{U(\bar{v})\}$, which is satisfied by the same values of $\bar{v}$ that satisfy the first-order formula U. The above formula is true in I iff for all domain values $\bar{v}$ satisfying $U(\bar{v})$, there is a computation of π starting in the state $\bar{x}{=}\bar{v}$ that halts in a state such that $V(\bar{x})$ is true. Thus if π is deterministic, the formula expresses total correctness. It is well known that there cannot be a sound and relatively complete axiom system based on a first-order oracle for total correctness assertions [Ap81]. Thus there cannot be a sound and relatively complete axiom system based on a first-order oracle for the full logic.

Moreover, it is shown in [Gr85] that there cannot be an axiom system that is sound and relatively complete for total correctness even if the axiom system is required to be sound only for expressive interpretations. This result requires a much deeper analysis than the one in [Ap81]. We simply note that the result of [Gr85] implies there cannot be a relatively complete axiom system for our full logic, even with soundness restricted to expressive interpretations.

9. Possible Extensions Beyond L4

It appears that the construction of S* can be extended to work for programs for which there is a bound on the number of variables that can be accessed by any statement. Here is a brief sketch of how this might be done. Suppose that no statement in π can access more than k variables, for some k. Each of the procedures in π^* will have, in addition to the formal parameters given in the translation of L4 programs, a list of k new ground parameters. These parameters will be used to pass along variables that can be accessed as global variables by procedures in the original program. Closures would be extended to include an encoding of the correspondence between the global variables that can be accessed by the procedure of the original

program, and the list of k new variables in the simulation. It would be necessary to generalize Lemma 1 to show that a finite encoding could still be used in this case. These ideas suggest the possibility of finding a sound and relatively complete axiom system for a language with restricted, but useful, access and update of global variables. The assertion language would need to have a more general way of referring to the values of global variables. For instance, *locations* are used for this purpose in [THM83, HMT84].

10. Conclusions

Although the sublanguage L4 of Algol has been extensively studied, the results of this paper are the first to demonstrate that a syntactic axiom system for L4 is sound and relative complete in the sense of Cook. One essential property of a "syntactic axiom system" is that it should allow assertions about a compound statement to be deduced from assertions about the component statements. Intuitively, this is difficult in L4 because of the infinite range problem. Although the completeness proof contains much technical detail, the axiom system itself is very simple and natural to use. The use of the axioms was illustrated with an example.

Compared to previous axiom systems for languages with finite range, such as those in [Co78, Go75, Cl79], the main new elements of our axiom system for infinite range are the arbitrary nesting of formulas with $\rightarrow$, quantification over procedures and ground variables, and the recursion rule for higher-typed procedures. What has not changed in comparison with the earlier axiom systems is the style of reasoning: proofs in our axiom system have a very simple, direct relationship to the syntax of the programs. Such simplicity is a key to the practical usefulness of axiom systems in the style of Hoare. Because the axioms are closely related to the syntax of programs, they provide direct intuition into the meaning of programming language constructs, and can be used for teaching programming. Two other important advantages of this transparent style of reasoning are that it facilitates the development of programs together with their proofs, and that programmers can carry out proofs with less than full formality by focussing on the most important assertions.

Our approach allows nondeterminism to be handled easily; one axiom is needed for nondeterminism, and our approach to proving relative completeness handles nondeterminism with no difficulty.

It is appropriate to comment on some of the technical difficulties present in the completeness proof, and prospects for their removal. We originally studied the relative completeness of the axioms in expressive, *Herbrand definable* interpretations [GCH83].

Using the property of Herbrand definability, one can write a program that, in one execution, enumerates all the values in the domain of the interpretation. This enumeration can be used to construct a simulation of L4 programs that has cleaner semantics than the simulation used here.

First, we can use a simpler mapping between domain elements and natural numbers. Bounded Herbrand definable interpretations are finite; for such interpretations we can use a bijection between domain elements and an initial segment of the natural numbers. In unbounded Herbrand definable interpretations, the order in which an enumeration program lists the domain elements gives a bijection between the domain elements and the natural numbers. In contrast, in the non-Herbrand case, it is necessary to add the input parameter b to start the enumeration, and only a subset of k-tuples of domain elements can be used to represent natural numbers. In the Herbrand case, using the cleaner simulation, the formula corresponding to the strongest postcondition of a statement S with respect to a precondition P is simply SP[S*; P]; for the general case, we must use $\exists b$ SP[S*; $P \wedge W(\bar{r}^*)$].

It may be possible to formulate the proof in a way that partly suppresses this change. For instance, one could define SP*[S; P] to be $\exists b$ SP[S*; $P \wedge W(\bar{r}^*)$]. Then one would prove that SP* has such properties as SP*[(S1; S2); P] $\equiv$ SP*[S2; SP*[S1; P]]. It is here that one would appeal to technical facts such as that S* does not change the values of $\bar{r}^*$, and the property of the variable b given in Lemma 8. A potential problem with this approach is that some of the axioms have restrictions on the variables that can appear free in formulas. Recall that we often work with formulas in which only some subset of the $\bar{r}^*$ variables appear free and that is important to keep track of these subsets because of restrictions on the axioms. In our completeness proof, we have been careful to show that all the necessary restrictions are observed. However, changing to the SP* notation may tend to obscure which of the $\bar{r}^*$ variables appear free in a given formula, and thus may hide details of the proof. Our feeling is that for understanding the proof, it is better to keep such details in plain view.

A separate, and more serious, source of technical difficulty is that we must reason about procedure declarations that contain free procedure names. We saw in the example of Section 5 that this does not add complexity when reasoning about a particular program in our axiom system. However, in order to handle the general case for the completeness proof, we have to consider the general pattern of chains of procedure references. This leads to notational problems such as the need for $\bar{r}^f$, $\bar{r}^g$, $\bar{r}^{*f}$, and $\bar{r}^{*g}$. We feel that the technical details related to this are inherent in proving the completeness of an axiom system that reasons about L4 programs according to their

syntax.

A final area of difficulty is specializing a general assertion about a procedure $p(\bar{r}, \bar{x})$ in order to reason about a particular call $p(\bar{q}, \bar{y})$, with actual parameters. Here, much of the difficulty comes because $\bar{q}$ can refer to declared procedures having free procedure names in their declarations. This implies that in the general case we must deal again with chains of procedure references. Again we feel that much of this difficulty is inherent in the language L4, although perhaps with further study one could find different methods of presentation.

The idea of proving relative completeness using interpreter programs and a simulation may be applicable to other programming languages. It may also be useful for showing true relative completeness of other axioms systems for the language L4.

Although our relative completeness proof uses some techniques from previous papers on characterizing languages that can have sound and relative complete axiom systems [Li77, CGH83], the main results of those papers are of a quite different kind: They do not give syntactic axiom systems in which assertions for a complex program are deduced from assertions about its components. The notion of a programming language used in [Li77, CGH83], called an acceptable programming language, is too abstract to give syntactic reasoning systems. In particular, the notion of an acceptable programming language does not specify that the semantics of a compound program is related to the semantics of its parts. The relative completeness result of [Gr84/86] is also based on acceptable programming languages, and so the same comments apply.

A final conclusion is that the kind of analysis of the semantics of a programming language that we carried out in Part 1 is of interest independently from the main result, because it gives valuable insight into the power of language features and the effect of restrictions. This kind of detailed knowledge about semantics can be useful for suggesting other languages that may also be axiomatizable and in the design of new languages.

Acknowledgements

We wish to give special thanks to Albert Meyer, who showed a great interest in this work and very strongly encouraged us to present it in a complete form. Without this encouragement, the paper might not have been finished. Also, while the final version of the paper was being prepared, we had many interesting discussions with Albert Meyer and Kurt Sieber on issues related to L4 in a research seminar at M.I.T.

Appendix 1. Removing Aliasing from L4 Programs

In this appendix, we briefly sketch a method for removing aliasing from L4 programs. The basic idea is to replace each procedure declaration with a set of new declarations, where there is one new declaration corresponding to each case of aliasing. All procedure calls are modified to use one of the new procedures. The resulting procedure calls have no aliasing.

Consider a declaration $p(\bar{r}, \bar{x}) \leftarrow b$ and a partition Π of the variables in $\bar{x}$. Intuitively, we will declare a new procedure p_Π to handle calls of p with the pattern of aliasing given by Π. Let $\bar{x}'$ be a list of variables containing one variable from each equivalence class of Π. For convenience, let us assume that the variables in $\bar{x}'$ are listed in the same order that they appear in $\bar{x}$, so that $\bar{x}'$ is formed by deleting variables from $\bar{x}$. Moreover, let the representative element of each equivalence class be the variable in the equivalence class that appears leftmost in $\bar{x}$.

We will use $\bar{x}'$ as the list of formal ground parameters of p_Π. Intuitively, the body of p_Π is formed by copying the body of p under a substitution that maps each formal ground variable to its representative in the list $\bar{x}'$. So let $subst_\Pi$ be a substitution mapping each variable in $\bar{x}$ to its representative in $\bar{x}'$. When we copy simple statements such as an assignment statement, we simply apply $subst_\Pi$. Thus if p contains the statement $x := e$, the corresponding statement in p_Π will be $x(subst_\Pi) := e(subst_\Pi)$. (We may assume, without loss of generality, that the *local* variables in the body of p are distinct from $\bar{x}$, so that $subst_\Pi$ leaves them unchanged.)

To copy the procedure calls in the body of p involves a bit more than just a substitution. If a call $p'(\bar{s}, \bar{y})$ appears in the body of p, then we translate it into a call $p'_{\Pi'}(\bar{s}', \bar{y}')$. Here $p'_{\Pi'}$ is a procedure that handles calls to p' under the equivalence class Π'. The equivalence class Π' for this call depends on both Π and $\bar{y}$; that is, we form $\bar{y}(subst_\Pi)$, and then Π' is the partition of the formal ground parameters of p' that matches it. Next, the list $\bar{y}'$ is formed in the obvious way: we take $\bar{y}(subst_\Pi)$ and then delete repetitions from this list. (To be precise, since we defined the representative element of each equivalence class to be the variable appearing earliest in the formal list, we will form $\bar{y}'$ by keeping the first appearance of each variable in $\bar{y}(subst_\Pi)$ and deleting later appearances.)

Next we will explain how the list of procedure parameters $\bar{s}'$ is formed. We have replaced each declared procedure by a set of new procedures that handle the various cases of aliasing; we also make an analogous change in the formal procedures, replacing each formal procedure name by a list of new formal procedure names to handle aliasing

in calls of formal procedures. The details of this are straightforward, if tedious. As an example, consider a procedure p in the original program, whose type is $\tau = (\tau_1, \text{var}^n)$, where $\tau_1 = (\text{var}^m)$. By the previous discussion, we replace each procedure of type τ_1 by a set of procedures for the various partitions of m ground parameters; these procedures have types ranging from (var^1) to (var^m). Now, for each partition Π there will be a procedure p_Π. These procedures will have, in place of a single formal procedure name of type τ_1, a list of formal procedure names, with one new name for each case of aliasing of a procedure of type τ_1. It is straightforward to see, by induction on the structure of procedure types, that this construction assigns a new procedure type to each type used in the original program. Thus, returning to the translation of $p'(\bar{s}, \bar{y})$ into $p'_{\Pi'}(\bar{s}', \bar{y}')$, the list $\bar{s}'$ is formed from $\bar{s}$ by replacing each procedure name by a list of all the procedures needed to handle cases of aliasing of that procedure.

This completes our sketch of the essential ideas of the translation to remove aliasing.

References

[Ap81] K.R. Apt, "Ten years of Hoare's logic, a survey -- part I," *ACM Toplas*, 3, 431-483, 1981.

[Cl79] E.M. Clarke, Jr., "Programming language constructs for which it is impossible to obtain good Hoare-like axioms," *Journal of the ACM*, 26, 129-147, 1979.

[Cl84] E.M. Clarke, Jr., "The characterization problem for Hoare logics," *Phil. Trans. R. Soc. Lond.* A 312, 423-440 (1984). Also reprinted in *Mathematical Logic and Programming Languages*, C.A.R. Hoare and J.C. Shepherdson, eds., Prentice-Hall International Series in Computer Science, 1984.

[CGH83] E.M. Clarke, S.M. German, J.Y. Halpern, "Effective axiomatizations of Hoare logics," *Journal of the ACM*, 30, 612-636, 1983.

[Co78] S.A. Cook, "Soundness and completeness of an axiom system for program verification," *SIAM Journal on Computing*, 7:1, 70-90, February, 1978.

[DJ82/83] W. Damm, B. Josko, "A sound and relatively* complete Hoare-Logic for a language with higher type procedures," Tech. Rept. Bericht Nr. 77, Lehrstuhl fur Informatik II, RWTH Aachen, April, 1982; *Acta Informatica*, 20, 59-101, 1983.

[En72] H.B. Enderton, *A Mathematical Introduction to Logic*, Academic Press, New York, 1972.

[GCH83] S.M. German, E.M. Clarke, J.Y. Halpern, "Reasoning about procedures as parameters," in *Proceedings of Conference on Logics of Programs*, LNCS 164, 206-220, June, 1983.

[GCH86a] S.M. German, E.M. Clarke, J.Y. Halpern, "True relative completeness of an axiom system for the language L4," in *Proceedings of Symposium on Logic in Computer Science*, June, 1986.

[GCH86b] S.M. German, E.M. Clarke, J.Y. Halpern, "True relative completeness of an axiom system for the language L4," with full details of proofs, privately circulated, June, 1986.

[GH83] S.M. German, J.Y. Halpern, "On the power of the hypothesis of expressiveness," IBM Research Report RJ 4079, 1983.

[Go85] A. Goerdt, "A Hoare calculus for functions defined by recursion on higher types," in *Proceedings of Conference on Logics of Programs*, LNCS 193, 106-117, June, 1985.

[Go75] G.A. Gorelick, "A complete axiomatic system for proving assertions about recursive and non-recursive programs," Tech. report 75, Dept. of Computer Science, Univ. of Toronto, 1975.

[Gr84/86] M. Grabowski, "On relative completeness in programming logics," in *Proc. Eleventh Annual ACM Symposium on Principles of Programming Languages*, 258-261, January, 1984; "On relative completeness of Hoare logics," *Information and Control*, Vol. 66, Numbers 1/2, 29-44, July/August 1985.

[Gr85] M. Grabowski, "On the relative incompleteness of logics for total correctness," in *Proceedings of Conference on Logics of Programs*, LNCS 193, 118-127, June, 1985.

[Ha84] J.Y. Halpern, "A good Hoare axiom system for an Algol-like language," in *Proc. Eleventh Annual ACM Symposium on Principles of Programming Languages*, 262-271, January, 1984.

[HMT84] J.Y. Halpern, A.R. Meyer, B.A. Trakhtenbrot, "The semantics of local storage, or what makes the free-list free?" in *Proc. Eleventh Annual ACM Symposium on Principles of Programming Languages*, 245-257, January, 1984.

[Jo83] B. Josko, "On expressive interpretations of a Hoare-logic for a language with higher-type procedures," Tech. Rept. Bericht Nr. 88, Lehrstuhl fur Informatik II, RWTH Aachen, 1983.

[La82] H. Langmaack, On termination problems for finitely interpreted ALGOL-like programs, *Acta Informatica*, 18, 79-108, 1982.

[Li77] R.J. Lipton, "A necessary and sufficient condition for the existence of Hoare logics," in *Proc. of the 18th IEEE Symposium on Foundations of Computer Science*, 1-6, October, 1977.

[Ol81/83] E.R. Olderog, "A characterization of Hoare's logic for programs with Pascal-like procedures," in *Proceedings of 15th ACM Symposium on Theory of Computing*, 329-329, 1983.

[Ol81/84] E.R. Olderog, "Correctness of programs with PASCAL-like procedures without global variables," Tech. Rept. 8110, Institut fur Informatik und Prakt. Math., Christian-Albrechts-Universitat Kiel, Nov. 1981; *Theoretical Computer Science*, 30, 49-90, 1984.

[Si85] K. Sieber, "A partial correctness logic for procedures," in *Proceedings of Conference on Logics of Programs*, LNCS 193, 320-342, June, 1985.

[THM83] B.A. Trakhtenbrot, J.Y. Halpern, A.R. Meyer, "From denotational to

operational semantics for Algol-like languages: an overview," in *Proceedings of Conference on Logics of Programs*, LNCS 164, 474-500, June, 1983.

[Ur83] P. Urzyczyn, "A necessary and sufficient condition in order that a Herbrand interpretation is expressive relative to recursive programs," *Information and Control* 56, 212-219, 1983.

Index