An Abstract Programming System

David A. Plaisted*
Department of Computer Science
UNC Chapel Hill
Chapel Hill, NC 27599-3175
Phone: (919) 967-9238
Fax: (919) 962-1799
Email: plaisted@cs.unc.edu

November 10, 2002

Abstract

The system PL permits the translation of abstract proofs of program correctness into programs in a variety of programming languages. A programming language satisfying certain axioms may be the target of such a translation. The system PL also permits the construction and proof of correctness of programs in an abstract programming language, and permits the translation of these programs into correct programs in a variety of languages. The abstract programming language has an imperative style of programming with assignment statements and side-effects, to allow the efficient generation of code. The abstract programs may be written by humans and then translated, avoiding the need to write the same program repeatedly in different languages or even the same language. This system uses classical logic, is conceptually simple, and permits reasoning about nonterminating programs using Scott-Strachey style denotational semantics.

Keywords

Abstract programming, program generation, program verification

1 Introduction

The purpose of the system PL is to permit the construction of proofs that can be viewed as abstract programs and translated into correct programs in a variety of programming languages. The emphasis is not on the automatic construction of the proofs but on the process of translating them into programs in specific programming languages. A programming language must satisfy certain conditions in order to be the target of such a translation, and typical procedural, functional, and logic

*This research was partially supported by the National Science Foundation under grant CCR-9972118.

programming languages satisfy these conditions. This system uses classical logic and Scott-Strachey domain theory [Sto77]. This system therefore, in theory, permits reasoning about nonterminating programs and nondeterministic programs. The system may also be used to construct abstract programs without proving them correct, and these abstract programs can also be translated into a variety of programming languages.

For a given programming language L, the axioms $PL(L)$ describe the properties the language L must satisfy in order for PL proofs to be translatable into L. A programming language L is *PL-feasible* if it satisfies the axioms of $PL(L)$. (Actually, the set of PL-feasible languages may differ from one program to another, because some programs may require different precisions of floating point numbers or various sizes of character strings or integers that may not be available in all languages, et cetera.) $PL(L)$ permits the construction of proofs that a particular L program P satisfies a specification. The system PL^* permits the construction of abstract proofs that correspond to $PL(L)$ proofs and therefore to correct programs in any PL-feasible language L. For each PL-feasible language L, there is an effective function from proofs in PL^* to correct programs in L. This permits the translation of PL^* proofs into correct programs in any PL-feasible language. An abstract programming language L^* corresponds to the system PL^*, and there are likewise effective functions from correct L^* programs to correct programs in any PL-feasible language L. It is possible to prove the correctness of L^* programs using PL^* or instead to gain confidence in the reliability of L^* programs by testing or some other means; thus it is not necessary to prove correctness in order to translate L^* programs into PL-feasible languages. Note that PL is not concerned with the details of the semantics of PL-feasible languages L; it only requires that L satisfy the axioms given for PL-feasible languages.

In order to have a realistic representation of algorithms at an abstract level in PL, it is necessary to include imperative features in PL that permit an accurate representation of operations and data structures that efficient algorithms use. For example, PL formalizes the destructive modification of data, as well as the side effects of operations on data; one cannot realistically describe quicksort without the former, and one cannot realistically describe binary tree manipulation routines without the latter. The operations in PL are few enough in number to preserve its simplicity, but inclusive enough to permit the generation of efficient code through the specification of destructive assignment statements and side effects.

2 Background and Discussion

There has been substantial work in program generation and programming logics. A number of papers discuss program synthesis based on constructive logic, type theory, and the Curry-Howard isomorphism [Con85, CS93]. For example, Martin-Löf's theory of types [ML82] and the calculus of constructions of Coquand and Huet [CH88, vHDRS95] are used for this purpose. Bittel [Bit92] and Kanovich [Kan91] describe program synthesis in intuitionistic logic. NuPRL [CAB+86] uses a hybrid system of logic and type theory. The proofs-as-programs paradigm of Bates and Constable [BC85] interprets constructive proofs as executable programs using the Curry-Howard isomorphism. However, such proofs contains both computational content and correctness arguments; in order to obtain efficient code, it is useful to separate these, which is not always simple (Berger and Schwichtenberg [BS96]). Avellone et al [AFM99] discuss this separation in the context of a system for reasoning about abstract data types. They discuss program synthesis from constructive proofs, but in which abstract data types have classical semantics. Jeavons et al [JPCB00] present another system for separating these parts of constructive proofs using the Curry-Howard isomorphism. Most such systems synthesize functional programs, but Mason [Mas97] studies the synthesis of imperative programs in a lambda calculus framework.

Another line of work in program generation makes use of *schemata*. The idea of schema based program synthesis [FLOR99] is to consider a schema as a generalized program that can be instantiated to a number of specific programs. Such schemata can be derived formally or constructed manually, and the correctness proofs can be manual or with automated assistance. Once obtained, schemata can be instantiated and combined to produce a variety of correct programs. Also, systematic methods exist for transforming schemata into different schemata. Huet and Lang [HL78] studied the transformation of one schema to another, and many other such transformation systems ([Par90]) have been studied. This can include tail recursion elimination, for example. An advantage of the schema-based approach to program generation as compared to constructive derivations of programs from scratch is that the deductive and programming tasks are easier. The schema-based approach approach also permits the use of classical logic in reasoning about schemata. Of course, schema-based development is also possible in constructive logics. Anderson and Basin [AB00] mention that schemata are not general enough to capture some programming knowledge, including the design patters of Gamma et al [GHJV95].

Schema based program synthesis as in Flener et al [FLOR99] is concerned with synthesizing a variety of logic programs, possibly in a single language, from a schema. A schema is typically *open*, which means that some of the predicates (representing procedures) are not defined. Thus, the schema can instantiate to different programs if different definitions of the undefined procedures are given. Flener et al [FLOR99] separate a schema into a *template*, which is an abstract program, and a *specification framework*, or collection of axioms giving the intended semantics of the problem domain. Their work was strongly influenced by the work of Smith [Smi90] in this respect. The semantics can either be *isoinitial*, a restriction of initial algebra semantics in which negative as well as positive equations are preserved [FLO98], or can be based on logic program semantics using *completions* of a logic program [LOT99]. A schema that is *steadfast* is guaranteed always to instantiate to correct programs. The synthesis process combines schemata, often by instantiating the open predicates of one schema using predicates from another. Büyükyildiz and Flener [BF97] study rules for the transformation of one logic program schema into another. Lau, Ornaghi, and Tärnlund [LOT99] discuss the relationship of schemata to object-oriented programming. Deville and Lau [DL94] discuss constructive, deductive, and inductive synthesis of logic programs.

Anderson and Basin [AB00] show how to view program schemata as derived rules of inference in higher-order logic. This approach can encompass both functional and logic programming languages. Like the schema approach, this approach relies on classical logic and does not make initial algebra assumptions that are typical of abstract data type theory. The formalism of Anderson and Basin [AB00] makes use of *schema variables*, which can be replaced by arbitrary functions. Also, Anderson and Basin [AB00] emphasize logic programs, but remark that schema based development also applies readily to functional programs. Shankar [Sha96] and Dold [Dol95] study program transformation in higher-order logic in the PVS system.

Manna and Waldinger's deductive tableau system [MW92] also uses classical logic for synthesis of functional programs. Ayari and Basin[AB01] show how to express this system in Isabelle using higher-order logic and higher-order resolution. They give an example of synthesizing sorting programs (including the quicksort program) in a functional language.

The common language runtime [Box02] of Microsoft is an attempt to ensure compatibility between different programming languages by compiling them all into a common intermediate language. This permits programs in different languages to communicate with each other.

PL has some features in common with the preceding systems. As with schemata, PL is based on classical logic, is not based on the Curry-Howard isomorphism, and is only concerned with type theory in an incidental way. PL also explicitly separates correctness arguments from the computational content of a program. Abstract PL programs are similar to schemata, or templates. An abstract program in PL is only partially specified, in the sense that some of the procedures it uses may not be defined. This corresponds to the *open* programs of Flener et al [FLOR99]. Steadfast logic programs correspond to a PL program fragment that satisfies a specification. Abstract programs can be combined in PL, as the schemata of Flener et al [FLOR99], by instantiating the undefined procedures of one program to the procedures of another and combining the programs. Thus an abstract PL program can be viewed as a rule of inference for constructing programs, as in the system of Anderson and Basin[AB00]. The goal of PL is to avoid the need to write the same program over and over in different languages by permitting abstract PL programs to translate to many languages. Also, PL even avoids the need to write the same program many times in the same language, because it permits substitutions for the names of the procedures in an abstract program.

However, the preceding approaches have a different emphasis than PL. PL does not emphasize the underlying logic; any sufficiently expressive logic, such as some version of set theory or higher-order logic, would suffice. In PL, a single logical formula mentions both an abstract formula and the properties it is assumed to have, rather than separating this information as is often done with schemata. In this, our approach is similar to that of Anderson and Basin [AB00]. Furthermore, PL semantics is based on Scott-Strachey style [Sto77] denotational semantics, and therefore can potentially reason about nonterminating and even nondeterministic computation. PL semantics is defined axiomatically, instead of by initial or iso-initial models. Also, the present paper is not concerned with program generation methodology *per se*, as are a number of other works. In PL, the emphasis is not on synthesis but on translation of an abstract program into efficient programs in a variety of languages. This is related to the research topic mentioned in Anderson and Basin [AB00] of developing a metatheory to transfer schema results from one area to another. Some other systems handle synthesis in particular languages including UNIX, object code, and logic programs. For example, Sanella and Tarlecki [ST89] discuss the formal development of ML programs from algebraic specifications. Bhansali and Harandi [BH93] discuss the synthesis of UNIX programs. Benini [Ben00] discusses program synthesis of object code. The process of translation of PL programs into other languages is automatic and does not require any planning or reasoning. In contrast to other systems, PL does not emphasize the transformation of one schema to another, except for the translation of schemata into specific languages and the combination of existing schemata. Another difference between PL and other approaches is that PL variables can only be replaced by procedure names, and not by arbitrary functions as in Anderson and Basin[AB00].

In addition, PL gives substantial attention to imperative features such as assignment statements and side effects that are important for efficient code generation. The treatment of side effects is somewhat similar to that of Mason [Mas97]. The example of quicksort from Ayari and Basin [AB01] uses a functional notation, in which array segments are concatenated, instead of the usual, more efficient approach of in-place processing of subarrays in the recursive step. Bornat [Bor00] gives a way to prove properties of pointer programs using Hoare logic. Another approach to side effects is given by Harman et al [HHZM01] and involves transforming programs to remove side effects. Though

PL emphasizes imperative languages, it also has applications to logic and functional programs. By comparision, few if any of the preceding systems emphasize translation into a variety of languages, nor do most of them emphasize imperative features of languages. In addition, the focus of many of these system is the method of program generation.

The focus of PL differs from that of the common language runtime, as well. The latter permits programs in different languages to communicate. PL permits an abstract program to translate into a variety of other languages at the source level, and thereby avoids the need to write the same program many times in many languages.

In general, PL is not so much concerned with how programs are synthesized as with axiomatizing their correctness in an abstract setting, so as to guarantee the correctness of their translations into specific languages. The user would typically write programs in PL and provide proofs of their correctness.

Current program generation methods have several problems: 1. The demands on the formal reasoning part of the process are too stringent. 2. The generated code is not always as efficient as possible. 3. The logic is often unfamiliar to the typical user.

The system PL seeks to overcome these problems by permitting the writing and debugging of abstract programs without any reasoning at all, if desired. However, it is possible to verify the abstract programs. These programs then translate into efficient code in a variety of other languages. This is possible because the abstract programs permit an imperative style of programming with assignment statements and side effects. The use of classical logic helps to solve the third problem.

3 Axioms of Program Language Semantics

3.1 Introduction to Axioms

The system $PL(L)$ refers to *program fragments* in programming languages L. A program fragment P of L is a portion of a program in L that specifies the definition of some procedures and data in terms of others. Data may be integers, arrays, lists, trees, or other data structures typically referenced by program variables. The *outputs* of P are the procedures and data that are defined in P in terms of other procedures and data. The *inputs* of P are the procedures and data that are referenced in P but not defined there. Thus if $\overline{x}$ are the inputs of P and $\overline{y}$ are the outputs of P, P defines a function from the semantics of $\overline{x}$ to the semantics of $\overline{y}$. $\overline{x}$ and $\overline{y}$ are *variables* of P in the system $PL(L)$.

There are several operations on program fragments in the system $PL(L)$. If P and Q are program fragments, then $P;Q$ represents the sequential composition of P and Q (P then Q). $PL(L)$ does not have a parallel composition operator; it would be possible to add additional operators such as parallel composition and object inheritance to $PL(L)$. $\exists xP$ represents P with the variable x declared "local" so that it is not visible outside of $\exists xP$. If Θ is a substitution, then $P\Theta$ is P with program variables (procedures and data) substituted as specified by Θ. μ is a least fixpoint operator on programs, corresponding to the definition of recursive procedures. $\uparrow^p_{\overline{x}} P$ represents P with the new procedure p defined; this corresponds to a program of the form **procedure** $p(\overline{x}); P$ having P as the procedure body. $\downarrow^p_{\overline{x}} P$ represents the application of a procedure to arguments. This corresponds to a program of the form P; **call** $p(\overline{x})$ where p is a procedure defined in P. The program P may be empty in this case. $x?P?Q$ represents the conditional "**if** x **then** P **else** Q" where P and Q are program fragments, possibly empty. $\mu(u, v, P, \geq)$ represents the "fixed point" of P with procedures u and v identified, where $\geq$ is the "definedness" ordering for the denotational semantics of P. Each program

P has a corresponding PL *textual syntax* P^{text} so that the textual syntax for $P;Q$ is $P^{text};Q^{text}$, the textual syntax for $\exists x P$ is **var** $x; P^{text}$, the textual syntax for $\uparrow^{p}_{\overline{x}} P$ is **proc** $p(\overline{x}); P^{text}$ **end** p, the textual syntax for $\downarrow^{p}_{\overline{x}} P$ is $P^{text};$ **call** $p(\overline{x})$, and the textual syntax for $x?P?Q$ is **if** x **then** P^{text} **else** Q^{text} **fi**. Also, $\mu(u, v, P(u, v), \geq)^{text}$ is typically $P(v, v)^{text}$. In addition to this, of course, an L program will have a syntax specified by the language L.

3.1.1 Side effects

In a realistic system, one needs to formalize imperative operations on data for efficiency; for example, one may have a program that repeatedly updates a database, or repeatedly modifies an array, graph or buffer. Creating a new copy of a data structure each time it is modified is inefficient. Typically one can assign values repeatedly to a program variable using assignment statements. However, procedures typically have only one definition. In order to accommodate this distinction in PL, there are both *procedure* and *data* variables, and the semantics of a data variable x in a program fragment P is an ordered pair (α, β) where α is the initial value of x (when P begins) and β is the final value of x (when P ends). By convention, $\alpha = x^{init}$ and $\beta = x^{fin}$. In PL, a procedure $p(x, y)$ with semantics $y^{fin} = x^{init}$ can express an assignment statement $y := x$. There are no assignment statements *per se* in PL, and no arithmetic or Boolean operators. PL procedures with an appropriate specification represent such statements and operators. In the translation to an L program, such procedures would translate to the corresponding assignment statements and operators.

For some algorithms, such as binary tree manipulation routines, pointer manipulations are necessary. A proper treatment of pointers requires a modification to the semantics of variables. If a variable points to the root of a binary tree, then the semantics of the variable should include the whole tree. If a variable points to the root of a LISP list, the semantics of the variable should include the entire list. Therefore, the semantics of a variable needs to include all other values that may be reached from the variable by a sequence of pointers. This implies that there are side effects. If x points to a binary tree T having T' as a subtree, and y points to T', then a change to a substructure of y will change x as well. In general, any change to a pointer will affect any structure containing this pointer; this is the kind of side effect that PL can formalize.

Side effects can also occur if a procedure modifies variables declared outside the procedure body. The syntax of PL-feasible languages L prohibits this, to simplify reasoning about L programs. However, read and write statements modify input and output files and buffers, and therefore imply side effects to variables declared outside a procedure body. Such variables are also called *global variables* for the procedure. To handle this, PL-feasible languages may consider certain *state variables* such as the status of input and output files as implicit parameters of every procedure. A procedure may also modify global variables indirectly by side effects. This is difficult to detect syntactically. We assume that this cannot happen. There are sufficient conditions to prohibit such side effects, such as the condition that no pointer manipulations or array element assignments can precede procedure definitions.

In order to reason about the side effects of one actual parameter on another, we assume that parameters are passed by value when possible. For arrays and complex data structures such as lists and binary trees, parameters are passed by reference. The semantics of parameters passed by reference must include not only their value but also some information about the address at which they are stored, in order to determine the side effects of a change of one parameter on another.

3.1.2 Functional and Logic Programming Languages

Because PL permits an imperative style of programming, it may be difficult to encode abstract PL programs in pure functional and logic programming languages. However, many functional and logic programming languages have imperative features added for efficiency, facilitating the translation of PL programs into these languages. Some restrictions on PL programs may facilitate their translation into pure functional and logic programming languags. For example, if a PL program is written in a *single assignment* style, in which each data variable is assigned at most once, then the translation into functional and logic programming languages appears to be fairly direct. The single assignment style of programming also minimizes side effects. A more restrictive class of PL programs are those without any data variables, and these should be even easier to translate into pure functional and logic programming languages.

3.1.3 Substitutions

There are a number of axioms and rules of inference about substitutions in PL. These are necessary in order to reason about specific instances of general programs. Suppose that $P(x,y)$ and $Q(u,v)$ are program fragments in $PL(L)$. Suppose one has assertions $A(x,y)$ and $B(u,v)$ expressing the properties of P and Q. The program fragment $P(x,y);Q(y,v)$ expresses a sequential composition of P and Q, with the variable y of Q replacing the variable u. It is desirable to reason about the properties of this combined program fragment. It is plausible to assume that the assertion $A(x,y) \wedge B(y,v)$ would hold for the program fragment $P(x,y);Q(y,v)$. However, deriving this assumption requires axioms about how the assertions A and B behave under substitutions to the programs P and Q. Therefore PL contains a number of axioms about substitutions and their influence on program semantics. These axioms enable the derivation of properties of substitution instances of a general program from properties of the general program, and therefore facilitate the construction of programs in PL from general building blocks. PL restricts such reasoning to substitutions that do not identify output variables, because this assumption simplifies the axioms.

As an example where identifying output variables leads to unusual behavior of instances of a general program, consider the program $P(x,y)$ equal to $x := x+1; y := y+1$ and the assertion $A(x,y) \equiv (x^{fin} = x^{init} + 1 \wedge y^{fin} = y^{init} + 1)$. The program $P(x,x)$ is then $x := x+1; x := x+1$ and no longer satisfies the assertion $A(x,x) \equiv (x^{fin} = x^{init} + 1 \wedge x^{fin} = x^{init} + 1)$. Instead, $P(x,x)$ satisfies the assertion $x^{fin} = x^{init} + 2$.

Because PL is a general system for reasoning about programs in various languages L, it is necessary for PL formulas to refer to programs in L. PL views L programs simply as strings in a language, with certain program variables (names of procedures and data variables) replaced by PL variables in order to reason about instances of general programs.

3.2 Terminology

L is a programming language and P and Q denote programs or fragments of programs in L. These are sometimes written as P^L and Q^L to specify L. Programs in L are assumed to satisfy the axioms of the system $PL(L)$ given below.

In the notation $[\overline{x}]P(\overline{y})$, P is a program fragment containing variables $\overline{y}$ that may represent procedures or data. Variables may appear more than once in $\overline{x}$ and $\overline{y}$. The variables $\overline{y}$ are the "schema variables" of program P and $\overline{x}$ is a listing of these variables in the order they will appear in assertions about P. The *free variables* $FV(P)$ of a program P are those procedures and variables of P that are not locally bound in P, so that they are available outside of P. Other variables of P are *bound*. By convention all free schema variables in P must appear in $\overline{x}$. Variables may appear in $\overline{x}$ that do not appear free in P. Such variables are also elements of $FV(P)$, by convention. $P(\overline{x})$ may be an abbreviation for $[\overline{x}]P(\overline{x})$. The *side effect variable* ψ is included as the last element in the list $\overline{x}$ even though ψ does not occur in P; this variable ψ is useful for reasoning about side effects. The variable ψ is a data variable, and the sort of ψ is the union of the sorts of all data variables. The semantics of ψ consists of pairs of the form (α, β) indicating that if α is the value of ψ at the beginning of the execution of P, then side effects of P cause β to be the value of ψ at the end of the execution of P. For example, if P changes a pointer at address a to point to b, then the semantics of the side effect variable ψ for P would consist of pairs (x_1, x_2) such that $x_2 = x_1$ if x_1 is a structure not containing the pointer a, and if x_1 does contain the pointer a, then x_2 would be x_1 with this pointer modified to point to b.

The side effect variable interacts with the sequential composition operator. The composition $P;Q$ of two program fragments has the precondition that $FV(P) = FV(Q)$. This precondition enables reasoning about side effects. For programs without side effects, this condition can be relaxed. Suppose $P(x)$ has only the free data variable x and $Q(y)$ has only the free data variable y. In order to compose P and Q, it is necessary to add y as a free variable to P and x as a free variable to Q. The *new variable rule* permits this, but requires the semantics of these new variables to reflect the side effects of the executions of P and Q. Thus in $P(x)$, the new variable y has semantics reflecting the side effects of the execution of P on y. Similarly, in $Q(y)$, the new variable x has semantics reflecting the side effects of the execution of Q on x. Therefore in the program fragment $P(x); Q(y)$, the overall semantics of x would reflect the effect of executing P, followed by the side effects of the execution of Q on x, and the semantis of y would reflect the side effects of executing P, followed by the effect of executing Q.

P^{in} denotes the set of *input variables* of program P and P^{out} denotes the *output variables*. Input variables of P are those that are free in P but not defined there. Output variables of P are variables that are defined in P and may or may not be used in P. Each variable may be *data*, which must be defined before it is used, or a *procedure*, which can be defined after it is used. No procedure variable may be defined twice. P^{proc} denotes the free variables of P that are procedures and P^{data} denotes those that are data. If $p \in P^{proc}$ then p takes zero or more arguments, which need not listed be among the variables of P and may either be procedures or data, and which may be inputs or outputs of p. The number of arguments of p is its *arity*. The notations p^{in}, p^{out}, p^{proc}, and p^{data}, each of which denotes a subset of $\{1, \ldots, n\}$ if n is the arity of p, indicate which arguments of p are inputs, outputs, procedures, and data. Similarly, $p^{i,in}$ et cetera give information about the i^{th} argument of p, and $p^{i,j,in}$ et cetera give information about the j^{th} argument of the i^{th} argument of p, if p is a procedure. In general, one writes $p^{\alpha,in}$ et cetera where α is a sequence of integers. To avoid dealing with sets of integers, one writes $p(\overline{x})^{in}$, defined as $\{x_i : i \in p^{in}\}$, et cetera. Let x^{Type} be the *type* of x, which we define as the function from α to the 4-tuple $(x^{\alpha,in}, x^{\alpha,out}, x^{\alpha,data}, x^{\alpha,proc})$, for integer sequences α. Let $x^{\alpha,Type}$ be the function from β to the 4-tuple $(x^{\alpha\beta,in}, x^{\alpha\beta,out}, x^{\alpha\beta,data}, x^{\alpha\beta,proc})$, for integer sequences α and β. Thus $x^{\alpha,Type}$ is the type of x^{α}, in a sense. Also, x_P or $x[P]$ denotes the variable x of the program P.

The notation $\{\overline{z} : [\overline{x}]P(\overline{y})\}$ means that $\overline{z}$ is a sequence of values representing possible semantics of the schema variables $\overline{x}$ that appear in P, z_i being the semantics of x_i, and $\overline{y}$ is the variables $\overline{x}$ listed in a possibly different order. Note that $\overline{z}$ is not a function of P, because P may be just a fragment of a larger program P', and some of the program variables in P may be procedures that are defined elsewhere in P'. The constraint $C^L_{[\overline{x}]P}$ represents the constraint on the semantics of $\overline{x}$ imposed by the program fragment P^L. Thus $C^L_{[\overline{x}]P}(\overline{y})$ denotes that $\overline{y}$ are possible semantics for the schema variables $\overline{x}$ that are consistent with the program fragment P^L. This is simply another notation for $\{\overline{y} : [\overline{x}]P^L\}$.

If x and y are variables or terms, $x \equiv y$ means that x and y are syntactically identical. If $\overline{x}$ and $\overline{y}$ are sequences of variables, then $\overline{x} \circ \overline{y}$ denotes the concatenation of these two sequences. Often this is written with a comma as $\overline{x}, \overline{y}$. If $\overline{x}$ is the sequence $x_1, \ldots, x_n$ of variables then $\overline{\{\overline{x}\}}$ denotes the set $\{x_1, \ldots, x_n\}$. The notation $\overline{x} \to \overline{y}$ indicates that $y_i \equiv y_j$ if $x_i \equiv x_j$. A *variable substitution* is a function from program variables to program variables, often indicated by Θ. If P is a program fragment and Θ is a variable substitution, then $P\Theta$ denotes P with free variables replaced as specified by Θ, and bound variables renamed to avoid captures. A variable substitution Θ is *output-injective* on P if for all distinct variables $x, y \in P^{out}$, $x\Theta \not\equiv y\Theta$. If Θ is output injective on P then $P\Theta^{out} = P^{out}\Theta$ and $P\Theta^{in} = P^{in}\Theta - P\Theta^{out}$. Thus a variable that is the image of both an input and an output variable, is an output variable. A variable substitution may only identify variables of the same type, both of which are either procedure variables or data variables.

The symbol Θ typically denotes a variable substitution and σ typically denotes a function from integers to integers. If Θ is one to one it is called a *variable renaming* and if σ is also one to one it is called an *integer permutation.* If $\overline{x}$ is a tuple of program variables and Θ is a variable substitution then $\overline{x}\Theta$ denotes $x_1\Theta, \ldots, x_n\Theta$. If σ is an integer function and $\overline{x}$ is any tuple, then $\overline{\sigma}(\overline{x})$ denotes $x_{\sigma(1)}, \ldots, x_{\sigma(n)}$. Also, if σ is an integer function and $\overline{x}$ is a tuple of program variables then $\hat{\sigma}(\overline{x})$ denotes the variable substitution such that $\overline{\sigma}(\overline{x}) = \overline{x}\hat{\sigma}(\overline{x})$, that is, the substitution $\{x_1 \to x_{\sigma(1)}, \ldots, x_n \to x_{\sigma(n)}\}$. The side effect variable ψ is always the last element of the list $\overline{x}$ of variables, which means that no variable substitution or integer function can change this property. For example, for all variable substitutions Θ, $\psi\Theta = \psi$.

The symbols f and g typically denote functions from variables to their semantics. If f is a function on free variables of P then $\overline{f}(\overline{x})$ denotes $(f(x_1), \ldots, f(x_n))$.

The formula $\Sigma x A[x]$ means $A[x] \wedge A[y] \supset x = y$, that is, A is *exclusive* for x.

The axioms of $PL(L)$ are as follows:

3.3 Definitions

Definition of constraint C in terms of colon notation

$$C^L_{[\overline{x}]P}(\overline{y}) \equiv \{\overline{y} : [\overline{x}]P^L\} \tag{1}$$

Definition of R on programs If R is a relation on semantics of program variables then

$$R^L([\overline{x}]P) \equiv \forall \overline{y}(\{\overline{y} : [\overline{x}]P^L\} \supset R(\overline{y})) \tag{2}$$

$R^{L,\psi}$ explicitly considers the side effect variable ψ. $R^{L,\overline{\psi}}$ does not. If neither superscript appears, either meaning is possible.

3.4 Axioms about P^{in} and P^{out}

The general idea is that if a variable is an input variable in one part of a program and an output variable elsewhere, it is an output variable for the whole program.

If Θ is an output injective variable substitution then

$$(P(\overline{x})\Theta)^{in} = P(\overline{x})^{in}\Theta - P(\overline{x})^{out}\Theta \tag{3}$$

If Θ is an output injective variable substitution then

$$(P(\overline{x})\Theta)^{out} = P(\overline{x})^{out}\Theta \tag{4}$$

$$(P(\overline{x}); Q(\overline{y}))^{in} = P(\overline{x})^{in} \cup Q(\overline{y})^{in} - (P(\overline{x})^{out} \cup Q(\overline{y})^{out}) \tag{5}$$

$$(P(\overline{x}); Q(\overline{y}))^{out} = P(\overline{x})^{out} \cup Q(\overline{y})^{out} \tag{6}$$

$$(u?P(\overline{x})?Q(\overline{y}))^{in} = P(\overline{x})^{in} \cup Q(\overline{y})^{in} \cup \{u\} - (P(\overline{x})^{out} \cup Q(\overline{y})^{out}) \tag{7}$$

$$(u?P(\overline{x})?Q(\overline{y}))^{out} = P(\overline{x})^{out} \cup Q(\overline{y})^{out} \tag{8}$$

$$\uparrow^{p}_{\overline{y}} P(\overline{x},\overline{y})^{in} = P(\overline{x},\overline{y})^{in} - \overline{\{\overline{y}\}} \tag{9}$$

$$\uparrow^{p}_{\overline{y}} P(\overline{x},\overline{y})^{out} = P(\overline{x},\overline{y})^{out} \cup \{p\} - \overline{\{\overline{y}\}} \tag{10}$$

$$\downarrow^{p}_{\overline{y}} P(\overline{x},p)^{in} = P(\overline{x},p)^{in} \cup p(\overline{y})^{in} - \downarrow^{p}_{\overline{y}} P(\overline{x},p)^{out} \tag{11}$$

$$\downarrow^{p}_{\overline{y}} P(\overline{x},p)^{out} = P(\overline{x},p)^{out} \cup p(\overline{y})^{out} \tag{12}$$

$$\exists w P^{in} = P^{in} \tag{13}$$

$$\exists w P^{out} = P^{out} - \{w\} \tag{14}$$

$$\mu(u,v,P,\geq)^{in} = P^{in} - \{u\} \tag{15}$$

$$\mu(u,v,P,\geq)^{out} = P^{out} \tag{16}$$

3.5 Axioms about free variables

$$FV(P) = P^{in} \cup P^{out} \tag{17}$$

3.5.1 Consequences of this axiom

$$FV(P;Q) = FV(P) \cup FV(Q) \tag{18}$$

$$FV(x?P?Q) = FV(P) \cup FV(Q) \cup \{x\} \tag{19}$$

$$FV(\uparrow^p_{\overline{y}} P(\overline{x}, \overline{y})) = \{p\} \cup \overline{\{\overline{x}\}} \tag{20}$$

$$FV(\downarrow^p_{\overline{y}} P(\overline{x}, p)) = \overline{\{\overline{x}\}} \cup \overline{\{\overline{y}\}} \cup \{p\} \tag{21}$$

$$FV(\exists z P) = FV(P) - \{z\} \tag{22}$$

$$FV(\mu(u, v, P(\overline{x}, u, v), \geq)) = \overline{\{\overline{x}\}} \cup \{v\} \tag{23}$$

3.6 Axioms about arguments to procedure variables

If $x \in FV(P)$ then

$$x^{Type}[P;Q] = x^{Type}[P] \tag{24}$$

If $x \in FV(Q)$ then

$$x^{Type}[P;Q] = x^{Type}[Q] \tag{25}$$

$$p^{i\alpha,Type}[\uparrow^p_{\overline{y}} P] = y_i^{\alpha,Type}[P] \tag{26}$$

If $x \in FV(\uparrow^p_{\overline{y}} P) - \{p\}$ then

$$x^{Type}[\uparrow^p_{\overline{y}} P] = x^{Type}[P] \tag{27}$$

$$y_i^{\alpha,Type}[\downarrow^p_{\overline{y}} P] = p^{i\alpha,Type}[P] \tag{28}$$

If $x \in FV(\downarrow^p_{\overline{y}} P) - \overline{\{\overline{y}\}}$ then

$$x^{Type}[\downarrow^p_{\overline{y}} P] = x^{Type}[P] \tag{29}$$

If $x \in FV(\exists y P) - \{y\}$ then

$$x^{Type}[\exists y P] = x^{Type}[P] \tag{30}$$

If $z \in FV(\mu(u, v, P, \geq)(\overline{x}, \overline{y}, v)) - \{y\}$ then

$$z^{Type}[\mu(u, v, P, \geq)(\overline{x}, \overline{y}, v)] = z^{Type}[P] \tag{31}$$

For any substitution Θ,

$$x\Theta^{Type}[P\Theta] = x^{Type}[P] \tag{32}$$

and

$$P\Theta^{proc} = P^{proc}\Theta \tag{33}$$

and

$$P\Theta^{data} = P^{data}\Theta \tag{34}$$

3.6.1 Consequences of the above

$$p^{in}(\overline{y})[\uparrow^{p}_{\overline{y}} P] = P^{in} \cap \overline{\{\overline{y}\}} \tag{35}$$

$$p^{out}(\overline{y})[\uparrow^{p}_{\overline{y}} P] = P^{out} \cap \overline{\{\overline{y}\}} \tag{36}$$

3.7 Axioms about program forming operations

Preconditions for the sequential composition operator The operation $P;Q$ is allowed if no procedure variable x is in $P^{out} \cap Q^{out}$ and if $P^{data} = Q^{data}$. The latter condition can be satisfied by adding extra variables to P and Q if necessary using the new variable axiom that appears below.

Definition of ϕ operator The definition of the sequential composition operator makes use of the ϕ operator, defined as follows: If f and g are semantic functions and P and Q are program fragments then the semantic function $\phi_{P,Q}(f,g)$ satisfies the following:

1. if $z \in P^{proc}$ then $\phi_{P,Q}(f,g)(z) = f(z)$.
2. if $z \in Q^{proc} - P^{proc}$ then $\phi_{P,Q}(f,g)(z) = g(z)$.
3. if $z \in P^{data} \cap Q^{data}$ and $f(z) = (\alpha, \beta)$ and $g(z) = (\beta, \gamma)$ then $\phi_{P,Q}(f,g)(z) = (\alpha, \gamma)$.

If f and g give a semantics for P and Q, then $\phi(f,g)$ gives a semantics for $P;Q$.

Preconditions for $\phi(f,g)$ operator $prec(\phi, P, Q, f, g)$ specifies

1. if $z \in P^{proc} \cap Q^{proc}$ then $f(z) = g(z)$.
2. if $z \in P^{data} \cap Q^{data}$ then $\exists \alpha\beta\gamma(f(z) = (\alpha, \beta) \wedge g(z) = (\beta, \gamma))$.

Sequential composition axiom

$$\{\overline{h}(\overline{x}\circ\overline{y}) : [\overline{x}, \overline{y}]P(\overline{x}); Q(\overline{y})\} \equiv \exists fg(\{\overline{f}(\overline{x}) : [\overline{x}]P(\overline{x})\} \wedge \{\overline{g}(\overline{y}) : [\overline{y}]Q(\overline{y})\}) \wedge h = \phi_{P,Q}(f,g) \wedge prec(\phi, P, Q, f, g). \tag{37}$$

Conditional axiom

$$\begin{aligned}\{w, \overline{u}, \overline{v} : [z, \overline{x}, \overline{y}]z?P(\overline{x})?Q(\overline{y})\} \quad \equiv \quad & (w = \textbf{true} \wedge \{\overline{u} : [\overline{x}]P(\overline{x})\}) \vee (w = \textbf{false} \wedge \{\overline{v} : [\overline{y}]Q(\overline{y})\}) \\ & \text{if } \overline{x} = FV(P) \wedge \overline{y} = FV(Q)\end{aligned} \tag{38}$$

Here P or Q may be empty.

Deleting output axiom Intuitively, this axiom declares z to be a local variable.

$$\forall \overline{x}'\overline{y}'(\exists z'\{\overline{x}', z', \overline{y}' : [\overline{x}, z, \overline{y}]P\} \equiv \{\overline{x}', \overline{y}' : [\overline{x}, \overline{y}]\exists z P\}) \tag{39}$$

Preconditions for procedure operator axiom The operation $\uparrow_{\overline{y}}^{p} P(\overline{x},\overline{y})$ is allowed if no variable x_i is in P^{out}, and if no execution of p can have side effects on the variables $\overline{x}$. That is, $R^L(P(\overline{x},\overline{y}))$ where $R(\overline{u},\overline{v}) \equiv \forall i(u_i^{init} = u_i^{fin})$.

Procedure operator axioms Intuitively, these axioms define a new procedure p having the formal parameters $\overline{y}$.

$$\forall \overline{u} \exists q \{\overline{u}, q : [\overline{x}, p] \uparrow_{\overline{y}}^{p} P(\overline{x},\overline{y})\} \tag{40}$$

$$\forall p q \overline{u}(\{\overline{u}, q : [\overline{x}, p] \uparrow_{\overline{y}}^{p} P(\overline{x},\overline{y})\} \supset \forall \overline{v}(\{\overline{u}, \overline{v} : [\overline{x}, \overline{y}] P(\overline{x},\overline{y})\} \equiv q(\overline{v}))) \tag{41}$$

where p is a new variable or an input variable of P.

Preconditions for application axiom The operation $\downarrow_{\overline{y}}^{p} P$ is allowed if $P; \downarrow_{\overline{y}}^{p}$ is allowed. The fragment P may be empty, in which case there are no preconditions and $\downarrow_{\overline{y}}^{p} P$ is equivalent to $\downarrow_{\overline{y}}^{p}$.

Application axioms Intuitively, this operator calls a procedure p with actual parameters $\overline{y}$.

$$\forall \overline{u} \overline{v} q(\{\overline{u}, \overline{v}, q : [\overline{x}, \overline{y}, p] \downarrow_{\overline{y}}^{p} P(\overline{x}, p)\} \equiv \{\overline{u}, \overline{v}, q : [\overline{x}, \overline{y}, p](P(\overline{x}, p); \downarrow_{\overline{y}}^{p})\}) \tag{42}$$

$$\forall \overline{v} q(\{\overline{v}, q : [\overline{y}, p] \downarrow_{\overline{y}}^{p}\} \equiv q(\overline{v})) \tag{43}$$

Least fixpoint axiom If

$$\forall \overline{x}' u' \Sigma \overline{y}' v' \{\overline{x}', u', \overline{y}, v' : P(\overline{x}, u, \overline{y}, v)\}$$

and $\overline{y}, v$ are outputs and $\overline{x}, u$ are inputs of P and u, v are procedure variables then

$$\begin{aligned} \{\overline{x}', \overline{y}', w' : \mu(u, v, P, \geq)(\overline{x}, \overline{y}, v)\} \quad &\equiv \\ (\{\overline{x}', w', \overline{y}', w' : P(\overline{x}, v, \overline{y}, v)\} \quad &\wedge \quad \forall z' \forall \overline{y}''(\{\overline{x}', z', \overline{y}'', z' : P(\overline{x}, u, \overline{y}, v)\} \supset z' \geq w')) \end{aligned} \tag{44}$$

3.8 Axioms about variables

Correspondence axiom This axiom implies that semantics for procedure variables not appearing free in P can be arbitrary. However, data variables, even not free in P, can be influenced by side effects of the execution of P. Note that $([\overline{x}]P)^{data}$ may include data variables in $\overline{x}$ that are not free in P.

If $\{(u_i, x_i) : x_i \in FV(P) \cup ([\overline{x}]P)^{data}\} = \{(v_j, y_j) : y_j \in FV(P) \cup ([\overline{y}]P)^{data}\}$ then

$$\{\overline{u} : [\overline{x}]P\} \supset \{\overline{v} : [\overline{y}]P\}. \tag{45}$$

Alternate version If $\forall u \in FV(P) \cup ([\overline{x}]P)^{data} \cup ([\overline{y}]P)^{data} f(u) = g(u)$ then

$$\{\overline{f}(\overline{x}) : [\overline{x}]P\} \supset \{\overline{g}(\overline{y}) : [\overline{y}]P\}. \tag{46}$$

New variable axiom If y is a data variable that does not appear free in P or in $\overline{x}$ then

$$\{\overline{u}, v, w : [\overline{x}, y, \psi]P\} \equiv \{\overline{u}, v : [\overline{x}, \psi]P\} \wedge \{\overline{u}, w : [\overline{x}, \psi]P\} \tag{47}$$

Variable renaming axiom If Θ is a variable renaming then

$$\{\overline{y} : [\overline{x}]P(\overline{x})\} \supset \{\overline{y} : [\overline{x}\Theta]P(\overline{x}\Theta)\} \tag{48}$$

Note that variable renamings are variable substitutions and are output injective.

Special case If σ is an integer permutation then

$$\{\overline{y} : [\overline{x}]P(\overline{x})\} \supset \{\overline{y} : [\overline{\sigma}(\overline{x})]P(\overline{\sigma}(\overline{x}))\} \tag{49}$$

Equality axiom

$$(\{\overline{y} : [\overline{x}]P\} \wedge x_i \equiv x_j) \supset y_i = y_j \tag{50}$$

Definitional independence axiom The idea of this axiom is that any semantics for an instance of a program must also satisfy the constraints of the general program, that is, the definitions are independent of the instance of the program.

If $\overline{x} \to \overline{y}$ and $P(\overline{y})$ does not identify distinct outputs or data variables of $P(\overline{x})$ then

$$\forall \overline{z}(\{\overline{z} : [\overline{y}]P(\overline{y})\} \supset \{\overline{z} : [\overline{x}]P(\overline{x})\}) \tag{51}$$

Alternative version If Θ is a variable substitution that is output-injective on P and does not identify two data variables then

$$\forall \overline{z}(\{\overline{z} : [\overline{x}\Theta]P(\overline{x}\Theta)\} \supset \{\overline{z} : [\overline{x}]P(\overline{x})\}) \tag{52}$$

Examples of definitional independence Let $P(x, y, u, v)$ be the program $y := x+1; v := u+1$. Consider the program $P(x, y, y, v)$ which is $y := x+1; v := y+1$. Then $\{((1,1), (0,2), (0,2), (0,3)) : P(x, y, y, v)\}$. Definitional independence asserts that $\{((1,1), (0,2), (0,2), (0,3)) : P(x, y, u, v)\}$ which is not correct because u is not modified in $P(x, y, u, v)$. The problem is that two data variables have been identified. However, there is a semantics for $P(x, y, u, v)$ in which the final values of the variables (x, y, u, v) are $(1, 2, 2, 3)$, respectively.

Definitional independence applies to recursive procedure definitions. For this example, assume that L has function procedures. Let $P(f, g, h, k)$ define $f(x)$ as "if $x = 0$ then 0 else $g(x) + k(h(x))$." Then $P(f, g, h, f)$ defines $f(x)$ as "if $x = 0$ then 0 else $g(x)+f(h(x))$." Any semantics for $P(f, g, h, f)$ is also a semantics for $P(f, g, h, k)$.

3.9 Consequences of the above axioms

Permutation axiom If σ is an integer permutation then

$$\{\overline{y} : [\overline{x}]P\} \supset \{\overline{\sigma}(\overline{y}) : [\overline{\sigma}(\overline{x})]P\} \tag{53}$$

Variable independence axiom If x_i is a procedure variable that does not appear free in P then $\{\overline{y} : [\overline{x}]P\}$ does not depend on y_i, so

$$\{\overline{y} : [\overline{x}]P\} \supset \forall y_i \{\overline{y} : [\overline{x}]P\}. \tag{54}$$

4 Definitional Independence and Fixed Points

In order to justify the reasonableness of the preceding axioms, it is possible to show that programming languages with certain properties satisfy definitional independence and the fixpoint axiom. For the former, suppose that $\overline{y}$ and $\overline{z}$ are procedure variables of P and $\overline{w}$ are data variables. Let $\overline{w}^{init}$ be $(w_1^{init}, \ldots, w_n^{init})$ and let $\overline{w}^{fin}$ be $(w_1^{fin}, \ldots, w_n^{fin})$. Writing the assertion $\{(\overline{u}, \overline{v}, \overline{x}) : P(\overline{y}, \overline{z}, \overline{w})\}$ as $\{[\overline{u}, \overline{x}^{init} \to \overline{v}, \overline{x}^{fin}] : P[\overline{y}, \overline{w}^{init} \to \overline{z}, \overline{w}^{fin}]\}$ indicates that $\overline{y}$ are the input variables of P, $\overline{z}$ are the output variables, $\overline{u}$ are possible semantics of $\overline{y}$, $\overline{v}$ are possible semantics of $\overline{z}$, and $\overline{x}$ are possible semantics of $\overline{w}$. For simplicity, $\overline{w}$ is ignored from now on, because it does not affect the argument.

Definition 4.1 *A programming language L is* denotational *if for all programs P in L there is a monotonic and continuous functional τ_P such that $\{[\overline{u} \to \overline{v}] : P[\overline{y} \to \overline{z}]\}$ iff $\overline{v} = \tau_P(\overline{u})$ and $(\overline{y}, \overline{z}) \to (\overline{u}, \overline{v})$. (Here it is assumed that $\overline{y}$ and $\overline{z}$ are disjoint.) Also, if Θ is an output injective variable substitution, then $\{[\overline{u}' \to \overline{v}'] : P[\overline{y}\Theta \to \overline{z}\Theta]\}$ iff $(\overline{u}', \overline{v}')$ is the minimal element of the set $\{(\overline{u}'', \overline{v}'') : \overline{v}'' = \tau_P(\overline{u}'')$ and $(\overline{y}\Theta, \overline{z}\Theta) \to (\overline{u}'', \overline{v}'')$ and $u_i'' = u_i'$ if $y_i\Theta \notin \overline{z}\Theta\}$.*

The idea is that if Θ identifies input and output variables of P, then these variables are considered as output variables and they need to be minimized subject to the equation $\overline{v}'' = \tau_P(\overline{u}'')$. The condition $(\overline{y}, \overline{z}) \to (\overline{u}, \overline{v})$ means that if two elements y_i and y_j are the same, their semantics must be the same, and similarly for elements of $\overline{z}$ and for common elements of $\overline{y}$ and $\overline{z}$.

This condition is reasonable; it states that any inputs to a partial program have definitions outside the partial program, so nothing can be assumed about their semantics. But any procedure that is an output of the partial program has a definition in the partial program, and therefore has the denotationally smallest semantics that satisfies the definition. For example, in the partial program $P(x, y \to z, w)$, the definitions of x and y occur outside of P but the definitions of z and w occur in P, so the semantics of z and w are constrained by the semantics of x and y and the definitions of z and w in terms of x and y. Now consider $P(x, y \to y, w)$. This is an instance of $P(x, y \to z, w)$. The procedure y now has a recursive definition, and receives the least possible semantics satisfying its definition. The definition of the procedure x occurs elsewhere, so that the semantics of x is arbitrary. But the semantics of x determines the semantics of y and w.

Theorem 1 *If L is denotational then every program P in L satisfies definitional independence.*

Proof: Suppose $\{[\overline{u}' \to \overline{v}'] : P[\overline{y}\Theta \to \overline{z}\Theta]\}$. First assume that $\overline{y}\Theta$ and $\overline{z}\Theta$ are disjoint. This means that no element of $\overline{y}\Theta$ is in $\overline{z}\Theta$. Since L is denotational, it must be that $(\overline{y}\Theta, \overline{z}\Theta) \to (\overline{u}', \overline{v}')$ and $\overline{v}' = \tau_P(\overline{u}')$ by definition 4.1. Since $(\overline{y}, \overline{z}) \to (\overline{y}\Theta, \overline{z}\Theta)$ and $(\overline{y}\Theta, \overline{z}\Theta) \to (\overline{u}', \overline{v}')$, $(\overline{y}, \overline{z}) \to (\overline{u}', \overline{v}')$ as well. Because $\overline{v}' = \tau_P(\overline{u}')$ and $(\overline{y}, \overline{z}) \to (\overline{u}', \overline{v}')$, $\{(\overline{u}', \overline{v}') : P(\overline{y}, \overline{z})\}$ also.

Now consider the case when $\overline{y}\Theta$ and $\overline{z}\Theta$ are not disjoint. Write $(\overline{y}\Theta, \overline{z}\Theta)$ as $(\overline{y}'\Theta, \overline{x}, \overline{x}, \overline{z}'\Theta)$, indicating by $\overline{x}$ the parts of $\overline{y}$ and $\overline{z}$ that are identified by Θ and by $\overline{y}'$ and $\overline{z}'$ the remaining parts of $\overline{y}$ and $\overline{z}$. Similarly, write $(\overline{u}', \overline{v}')$ as $(\overline{u}'', \overline{w}, \overline{w}, \overline{v}'')$. The idea of the definition is that $\overline{w}$

and $\overline{v}$ are chosen to be as small as possible subject to the condition that $(\overline{w}, \overline{v}'') = \tau_P(\overline{u}'', \overline{w})$, but $\overline{u}''$ is chosen to be equal to the corresponding components of $\overline{u}'$, which are not constrained.

Since L is denotational and $\{[\overline{u}' \to \overline{v}'] : P[\overline{y}\Theta \to \overline{z}\Theta]\}$, $(\overline{u}', \overline{v}')$ is the minimal element of the set $\{(\alpha, \beta) : \beta = \tau_P(\alpha)$ and $(\overline{y}\Theta, \overline{z}\Theta) \to (\alpha, \beta)$ and $\alpha_i = u'_i$ if $y_i\Theta \notin \overline{z}\Theta\}$.

From this it follows that $\overline{v}' = \tau_P(\overline{u}')$ and $(\overline{y}\Theta, \overline{z}\Theta) \to (\overline{u}', \overline{v}')$.

We need to show that $\{[\overline{u}' \to \overline{v}'] : P[\overline{y} \to \overline{z}]\}$, that is, $(\overline{u}', \overline{v}')$ is the minimal element of the set $\{(\alpha, \beta) : \beta = \tau_P(\alpha)$ and $(\overline{y}, \overline{z}) \to (\alpha, \beta)$ and $\alpha_i = u'_i$ for all $i\}$.

First, $\overline{v}' = \tau_P(\overline{u}')$ as noted above.

Second, $(\overline{y}, \overline{z}) \to (\overline{u}', \overline{v}')$ because $(\overline{y}, \overline{z}) \to (\overline{y}\Theta, \overline{z}\Theta)$ for any Θ and (as noted above) $(\overline{y}\Theta, \overline{z}\Theta) \to (\overline{u}', \overline{v}')$.

Finally, we need to show that if $\beta = \tau_P(\alpha)$ and $(\overline{y}, \overline{z}) \to (\alpha, \beta)$ and $\alpha_i = u'_i$ for all i then $\alpha \geq \overline{u}'$ and $\beta \geq \overline{v}'$. But if $\alpha_i = u'_i$ for all i then $\alpha = \overline{u}'$. Thus $\beta = \tau_P(\alpha) = \tau_P(\overline{u}') = \overline{v}'$. Thus $\alpha = \overline{u}'$ and $\beta = \overline{v}'$.

As an example where the theorem fails if Θ identifies two data variables, consider the program $[x, y]x := x + 1$ and its instance $[x, x]x := x + 1$. The latter has the semantics $((0, 1), (0, 1))$ but not the former, because the value of y may not change.

Because functional, logic, and procedural languages are denotational, with reasonable definitions of their semantics, it is reasonable to assume that all these languages also satisfy definitional independence, and that all the inference rules in PL apply to all such languages.

In practice, one may use PL without a formal proof that the languages L satisfy definitional independence, to obtain programs that may have added reliability even if there is no formal proof of correctness.

The denotational property also suffices to justify the least fixpoint axiom.

Theorem 2 *Suppose L is denotational. Then the least fixpoint axiom is satisfied if one lets $\mu(u, v, P, \geq)$ be $P\Theta$ where Θ maps u to v but leaves all other variables unchanged.*

Proof: We show the least fixpoint axiom, axiom 44, which is the following: If

$$\forall \overline{x}' u' \Sigma \overline{y}' v' \{\overline{x}', u', \overline{y}, v' : P(\overline{x}, u, \overline{y}, v)\}$$

and $\overline{y}, v$ are outputs and $\overline{x}, u$ are inputs of P then

$$\begin{aligned} \{\overline{x}', \overline{y}', w' : \mu(u, v, P, \geq)(\overline{x}, \overline{y}, v)\} \quad &\equiv \\ (\{\overline{x}', w', \overline{y}', w' : P(\overline{x}, v, \overline{y}, v)\} \quad \wedge \quad &\forall z' \forall \overline{y}''(\{\overline{x}', z', \overline{y}'', z' : P(\overline{x}, u, \overline{y}, v)\} \supset z' \geq w')) \end{aligned} \quad (55)$$

The hypothesis $\forall \overline{x}' u' \Sigma \overline{y}' v' \{\overline{x}', u', \overline{y}, v' : P(\overline{x}, u, \overline{y}, v)\}$ is satisfied because L is denotational. Suppose $\{\overline{x}', \overline{y}', w' : \mu(u, v, P, \geq)(\overline{x}, \overline{y}, v)\}$. Defining μ as in the theorem, and using the correspondence axiom, this is equivalent to $\{\overline{x}', w', \overline{y}', w' : P(\overline{x}, v, \overline{y}, v)\}$. For the remaining part, write $\{\overline{x}', w', \overline{y}', w' : P(\overline{x}, v, \overline{y}, v)\}$ as $\{[\overline{x}', w' \to \overline{y}', w'] : P[\overline{x}, v \to \overline{y}, v]\}$ and write $\{\overline{x}', z', \overline{y}'', z' : P(\overline{x}, u, \overline{y}, v)\}$ as $\{[\overline{x}', z' \to \overline{y}'', z'] : P[\overline{x}, u \to \overline{y}, v]\}$. Because L is denotational, $\{[\overline{x}', z' \to \overline{y}'', z'] : P[\overline{x}, u \to \overline{y}, v]\}$ iff $(\overline{x}', z', \overline{y}'', z')$ is the minimal element of the set $\{(\overline{x}'', z'', \overline{y}''', z'') : (\overline{x}'', z'') = \tau_P(\overline{y}''', z'')$ and $((\overline{x}, u), (\overline{y}, v)) \to ((\overline{x}'', z''), (\overline{y}''', z''))$ and $\overline{x}'' =$

$\overline{x}'$ and $z'' = z'$ (because $\overline{x}$ and $\overline{y}$ are assumed disjoint). The condition $((\overline{x}, u), (\overline{y}, v)) \rightarrow ((\overline{x}'', z''), (\overline{y}''', z''))$ is true because $\overline{x}, u, \overline{y}$, and v are pairwise disjoint. Thus the only constraint on $(\overline{x}', z', \overline{y}'', z')$ is that $(\overline{x}', z') = \tau_P(\overline{y}'', z')$. However, because Θ identifies u and v, the corresponding constraint on $(\overline{x}', w', \overline{y}')$ is that w' should be minimal satisfying $(\overline{x}', w') = \tau_P(\overline{y}', w')$. Therefore $z' \geq w'$ as specified above. The other direction follows by similar reasoning, because L is denotational.

5 Inference Rules

Section 3.7 contains axioms for program language semantics expressed in terms of the operator :. These axioms lead to *relational* inference rules for deriving assertions $R^L(P)$ where P is an L program and R is a relation on semantics of program variables, using the definition

$$R^L([\overline{x}]P) \equiv \forall \overline{y}(\{\overline{y} : [\overline{x}]P\} \supset R(\overline{y}))$$

If R is such a relation and σ is an integer function then $R\sigma$ denotes the relation such that $R\sigma(\overline{y}) \equiv R(\overline{\sigma}(\overline{y}))$, that is, $R\sigma(y_1, \ldots, y_n)$ iff $R(y_{\sigma(1)}, \ldots, y_{\sigma(n)})$.

The system *relational* $PL(L)$ consists of the following inference rules, which are consequences of the axioms given in section 3.7:

$$\frac{R_1^L(P), R_1 \supset R_2}{R_2^L(P)} \quad \text{Underlying logic rule}$$

$$\frac{R_k^L(P), k \text{ arbitrary}}{\forall k R_k^L(P)} \quad \text{Universal quantification rule}$$

Rules about variables

$$\frac{R^L([\overline{x}]P(\overline{x})), \Theta \text{ is a variable renaming}}{R^L([\overline{x}\Theta]P(\overline{x}\Theta))} \quad \text{Variable renaming rule}$$

$$\frac{R^{L,\overline{\psi}}([\overline{x}]P(\overline{x})), \sigma \text{ is an integer permutation}}{R\sigma^{L,\overline{\psi}}([\overline{\sigma}^{-1}(\overline{x})]P(\overline{x}))} \quad \text{Permutation rule 1}$$

$$\frac{R^L([\overline{x}]P(\overline{x})), \sigma \text{ is an integer permutation}}{R\sigma^L([\overline{x}]P(\overline{\sigma}(\overline{x})))} \quad \text{Permutation rule 2}$$

$$\frac{R^L(P(\overline{x})), x_i \equiv x_j, R_1(\overline{y}) \equiv (R(\overline{y}) \wedge y_i = y_j)}{R_1^L(P(\overline{x}))} \quad \text{Equality rule}$$

$$\frac{R^L([\overline{x}]P(\overline{x})), \sigma \text{ is an integer function such that } \overline{\sigma} \text{ does not identify data variables of } \overline{x}}{R\sigma^L([\overline{x}]P(\overline{\sigma}(\overline{x}))}\ \text{Substitution rule}$$

$$\frac{R^L([\overline{x}]P), \forall f, g(R(\overline{f}(\overline{x})) \wedge \forall u \in FV(P) \cup ([\overline{x}]P)^{data}(f(u) = g(u))) \supset R_1(\overline{g}(\overline{y}))}{R_1^L([\overline{y}]P)}\ \text{Correspondence rule}$$

$$\frac{R^L([\overline{x}]P(\overline{x})), \Theta \text{ output injective on } P \text{ and does not identify data variables of } P}{R^L([\overline{x}\Theta]P(\overline{x}\Theta))}\ \text{Definitional independence rule}$$

$$\frac{R^{L,\psi}L([\overline{x}, \psi]P), y \text{ a new data variable}, R(\overline{u}, v) \wedge R(\overline{u}, w) \supset R_1(\overline{u}, v, w)}{R_1^{L,\psi}([\overline{x}, y, \psi]P)}\ \text{New variable rule}$$

Rules about program operations

$$\frac{R_1^L(P_1(\overline{y})), R_2^L(P_2(\overline{z})), prec(\phi, P, Q, f, g), \forall \overline{f}\overline{g}(R_1(\overline{f}(\overline{y})) \wedge R_2(\overline{g}(\overline{z})) \supset R(\phi_{P,Q}(\overline{f}, \overline{g})(\overline{y} \circ \overline{z})))}{R^L([\overline{y}, \overline{z}]P_1(\overline{y}); P_2(\overline{z}))}\ \text{Composition rule}$$

$$\frac{\begin{array}{c} R_1^L(P_1(\overline{y})), R_2^L(P_2(\overline{z})), \overline{y} \subseteq FV(P_1), \overline{z} \subseteq FV(P_2), \\ \forall x'\overline{y}'\overline{z}'(x' \wedge R_1(\overline{y}')) \vee (\neg x' \wedge R_2(\overline{z}')) \supset R(x', \overline{y}', \overline{z}') \end{array}}{R^L([x, \overline{y}, \overline{z}]x?P_1(\overline{y})?P_2(\overline{z}))}\ \text{Conditional rule}$$

$$\frac{R^L([\overline{u}, x, \overline{v}]P), R_1(\overline{u}, \overline{v}) \equiv \exists x R(\overline{u}, x, \overline{v})}{R_1^L([\overline{u}, \overline{v}]\exists x P)}\ \text{Output deletion rule}$$

$$\frac{R^L(P(\overline{x}, \overline{y})), R_1(\overline{u}, q) \equiv \forall \overline{v}(q(\overline{v}) \supset R(\overline{u}, \overline{v}))}{\forall \overline{u} \exists q R_1(\overline{u}, q) \wedge R_1^L([\overline{x}, p] \uparrow_y^p P(\overline{x}, \overline{y}))}\ \text{Procedure rule}$$

$$\frac{R^L(P(\overline{x}, p); \downarrow_y^p), P \text{ non-empty}}{R^L(\downarrow_y^p P(\overline{x}, p))}\ \text{Application rule 1}$$

$$\frac{q(\overline{v}) \supset R(\overline{v}, q)}{R^L([\overline{y}, p] \downarrow_y^p)}\ \text{Application rule 2}$$

$$\frac{\begin{array}{c} \forall \overline{x}'u'\Sigma\overline{y}'v'R(\overline{x}', u', \overline{y}', v'), R^L(P(\overline{x}, u, \overline{y}, v)), \overline{x}, u \in P^{in}, \overline{y}, v \in P^{out} \\ \forall \overline{x}'u'\overline{y}'v'(R(\overline{x}', u', \overline{y}', v') \supset \exists \overline{y}''v''\{\overline{x}', u', \overline{y}'', v'' : P(\overline{x}, u, \overline{y}, v)\}) \\ \forall \overline{x}'\overline{y}'w'(R_1(\overline{x}', \overline{y}', w') \equiv (R(\overline{x}', w', \overline{y}', w') \wedge \forall z' \forall \overline{y}''(R(\overline{x}', z', \overline{y}'', z') \supset z' \geq_L w'))) \end{array}}{R_1^L(\mu(u, v, P(\overline{x}, u, \overline{y}, v), \geq_L))}\ \text{Least fixpoint rule}$$

The second line of the hypothesis states that R does not hold on "bad inputs" to P, that is, inputs for which there is no output. The ordering $\geq_L$ depends on L and expresses the effect of recursion in L. Usually $\geq$ abbreviates $\geq_L$.

Theorem 3 . *These inference rules are logical consequences of the axioms for program language semantics which appear in section 3.7.*

Proof: The proofs for each rule follow:

Underlying logic rule Suppose $R_1^L(P)$ and $\forall \overline{y}(R_1(\overline{y}) \supset R_2(\overline{y}))$. By the definition of $R_1(P)$, axiom 2, $\forall \overline{y}(\{\overline{y} : [\overline{x}]P\} \supset R_1(\overline{y}))$. Since $\forall \overline{y}(R_1(\overline{y}) \supset R_2(\overline{y}))$, $\forall \overline{y}(\{\overline{y} : [\overline{x}]P\} \supset R_2(\overline{y}))$. Again by axiom 2, $R_2^L(P)$.

Universal quantification rule Suppose $R_k^L(P)$ for arbitrary k. By the definition of R_k^L, axiom 2, $\forall \overline{y}(\{\overline{y} : [\overline{x}]P\} \supset R_k^L(\overline{y}))$. Because k is arbitrary, $\forall k(\forall \overline{y}(\{\overline{y} : [\overline{x}]P\} \supset R_k^L(\overline{y})))$. Because k does not appear in the antecedent, $\forall \overline{y}(\{\overline{y} : [\overline{x}]P\} \supset \forall k R_k^L(\overline{y}))$. By axiom 2, $\forall k R_k^L(P)$.

Variable renaming rule Suppose $R^L([\overline{x}]P(\overline{x}))$ and Θ is a variable renaming. Suppose also that $\{\overline{y} : [\overline{x}\Theta]P(\overline{x}\Theta)\}$. By the variable renaming axiom, $\{\overline{y} : [\overline{u}]P(\overline{u})\} \supset \{\overline{y} : [\overline{u}\Theta^{-1}]P(\overline{u}\Theta^{-1})\}$ because Θ^{-1} is also a variable renaming. Letting $\overline{u}$ be $\overline{x}\Theta$, from the assumption $\{\overline{y} : [\overline{x}\Theta]P(\overline{x}\Theta)\}$ it follows that $\{\overline{y} : [\overline{x}]P(\overline{x})\}$. Since $R^L([\overline{x}]P(\overline{x}))$, $R(\overline{y})$. Therefore $\{\overline{y} : [\overline{x}\Theta]P(\overline{x}\Theta)\}$ implies $R(\overline{y})$. By the definition of R^L, $R^L([\overline{x}\Theta]P(\overline{x}\Theta))$.

Permutation rule 1 Suppose $R^L([\overline{x}]P(\overline{x}))$ and σ is an integer permutation. Suppose also that $\{\overline{z} : [\overline{\sigma}^{-1}(\overline{x})]P(\overline{x})\}$. By the permutation axiom, $\{\overline{\sigma}(\overline{z}) : [\overline{x}]P(\overline{x})\}$. Since $R^L([\overline{x}]P(\overline{x}))$, $R(\overline{\sigma}(\overline{z}))$ holds, or, $R\sigma(\overline{z})$. Thus $\{\overline{z} : [\overline{\sigma}^{-1}(\overline{x})]P(\overline{x})\}$ implies $R\sigma(\overline{z})$. Therefore $R\sigma^L([\overline{\sigma}^{-1}(\overline{x})]P(\overline{x}))$.

Permutation rule 2 Suppose $R^L([\overline{x}]P(\overline{x}))$ and σ is an integer permutation. By permutation rule 1, $R\sigma^L([\overline{\sigma}^{-1}(\overline{x})]P(\overline{x}))$. By the special case of the variable renaming axiom, $R\sigma^L([\overline{x}]P(\overline{\sigma}(\overline{x})))$.

Equality rule Suppose $R^L(P(\overline{x}))$, $x_i \equiv x_j$, and $R_1(\overline{y}) \equiv (R(\overline{y}) \wedge y_i = y_j)$. Recall that $P(\overline{x})$ abbreviates $[\overline{x}]P(\overline{x})$. Suppose also that $\{\overline{y} : [\overline{x}]P\}$. Since $R^L(P(\overline{x}))$, $R(\overline{y})$ holds. By the equality axiom, $(\{\overline{y} : [\overline{x}]P\} \wedge x_i \equiv x_j) \supset y_i = y_j$. Therefore $y_i = y_j$. Since $R_1(\overline{y}) \equiv (R(\overline{y}) \wedge y_i = y_j)$, $R_1(\overline{y})$ holds. Therefore $\{\overline{y} : [\overline{x}]P\}$ implies $R_1(\overline{y})$. Therefore $R_1^L(P(\overline{x}))$.

Substitution rule, special case Suppose $R^L([u, v, \overline{x}]P(u, v, \overline{x}))$ and σ is an integer function such that $\overline{\sigma}$ does not identify data variables of $\overline{x}$ and such that $\sigma(i) = i$ for $i \neq 2$ and $\sigma(2) = 1$. Then $R\sigma(u', v', \overline{x}') = R(u', u', \overline{x}')$. It is necessary to show $R\sigma^L([u, v, \overline{x}]P\overline{\sigma}(u, v, \overline{x}))$, that is, $R\sigma^L([u, v, \overline{x}]P(u, u, \overline{x}))$. Suppose $\{u', v', \overline{x}' : [u, v, \overline{x}]P(u, u, \overline{x})\}$. It is necessary to show $R\sigma(u', v', \overline{x}')$. By the correspondence axiom, $\{u', u', \overline{x}' : [u, u, \overline{x}]P(u, u, \overline{x})\}$ if σ is output injective. By definitional independence, $\{u', u', \overline{x}' : [u, v, \overline{x}]P(u, v, \overline{x})\}$ if u and v are not distinct data variables. Because $R^L([u, v, \overline{x}]P(u, v, \overline{x}))$, $R(u', u', \overline{x}')$. Therefore $R\sigma(u', v', \overline{x}')$. Hence $R\sigma^L([u, v, \overline{x}]P(u, u, \overline{x}))$. Therefore $R\sigma^L([u, v, \overline{x}]P\sigma(u, v, \overline{x}))$.

Substitution rule, general case Suppose $R^L([\overline{x}]P(\overline{x}))$ and σ is an integer permutation such that $\overline{\sigma}$ does not identify data variables of $\overline{x}$. Then $R\sigma^L([\overline{x}]P\sigma(\overline{x}))$ by combining permutation rule 1 and the variable renaming rule. Now, let σ be an arbitrary integer function from $\{1,\ldots,n\}$ to $\{1,\ldots,n\}$ where $\overline{x}$ has n components. Then σ can be expressed as the composition $\sigma_1\sigma_2\ldots\sigma_k$ where the σ_i are permutations and functions as in the preceding special case. For each such σ_i, $R\sigma_1\sigma_2\ldots\sigma_{i-1}^L([\overline{x}]P\sigma_1\sigma_2\ldots\sigma_{i-1}(\overline{x}))$ implies $R\sigma_1\sigma_2\ldots\sigma_i^L([\overline{x}]P\sigma_1\sigma_2\ldots\sigma_i(\overline{x}))$. By combining all these implications, $R^L([\overline{x}]P(\overline{x}))$ implies $R\sigma^L([\overline{x}]P\sigma(\overline{x}))$.

Correspondence rule Suppose $R^L([\overline{x}])P$. It is necessary to show $R_1^L([\overline{y}]P)$. For this, suppose $\{\overline{y}' : [\overline{y}]P\}$. Let f and g be such that $\overline{g}(\overline{y}) = \overline{y}'$ and $\forall u \in FV(P) \cup ([\overline{x}]P)^{data}(f(u) = g(u))$. Then $\{\overline{g}(\overline{y}) : [\overline{y}]P\}$. By the correspondence axiom, with f and g interchanged, $\{\overline{f}(\overline{x}) : [\overline{x}]P\}$. Because $R^L([\overline{x}]P)$, $R(\overline{f}(\overline{x}))$. By the definition of R_1, $R_1(\overline{g}(\overline{y}))$, that is, $R_1(\overline{y}')$.

Definitional independence rule Suppose $R^L([\overline{x}]P(\overline{x}))$ and Θ is output injective on P. Suppose also that $\{\overline{z} : [\overline{x}\Theta]P(\overline{x}\Theta)\}$. By the definitional independence axiom, if Θ is a variable substitution that is output-injective on P and does not identify two data variables, then $\forall\overline{z}(\{\overline{z} : [\overline{x}\Theta]P(\overline{x}\Theta)\} \supset \{\overline{z} : [\overline{x}]P(\overline{x})\})$. Therefore $\{\overline{z} : [\overline{x}]P(\overline{x})\}$. Since $R^L([\overline{x}]P(\overline{x}))$, $R(\overline{z})$ holds. Thus $\{\overline{z} : [\overline{x}\Theta]P(\overline{x}\Theta)\}$ implies $R(\overline{z})$. Therefore $R^L([\overline{x}\Theta]P(\overline{x}\Theta))$.

New variable rule Suppose $R^L([\overline{x},\psi]P)$, y is a new data variable for P, and $R(\overline{u},v) \wedge R(\overline{u},w) \supset R_1(\overline{u},v,w)$. Suppose also that $\{\overline{u},v,w : [\overline{x},y,\psi]P\}$. By the new variable axiom, $\{\overline{u},v : [\overline{x},\psi]P\}$ and $\{\overline{u},w : [\overline{x},\psi]P\}$. Because $R^L([\overline{x},\psi]P)$, $R(\overline{u},v)$ and $R(\overline{u},w)$. By the above implication, $R_1(\overline{u},v,w)$. Therefore $R_1^L([\overline{x},y,\psi]P)$.

Composition rule Suppose $R_1^L(P_1(\overline{y}))$, $R_2^L(P_2(\overline{z}))$, $prec(\phi,P_1,P_2,f,g)$, and $\forall\overline{f}\overline{g}(R_1(\overline{f}(\overline{y}))\wedge R_2(\overline{g}(\overline{z})) \supset R(\phi_{P_1,P_2}(\overline{f},\overline{g})(\overline{y}\circ\overline{z})))$. Suppose also that $\{\overline{y}'\circ\overline{z}' : P_1(\overline{y});P_2(\overline{z})\}$. Then there must be a semantic function h such that $\{\overline{h}(\overline{y}\circ\overline{z}) : P_1(\overline{y});P_2(\overline{z})\}$. By the sequential composition axiom 37, there are semantic functions f and g such that $\{\overline{f}(\overline{y}) : P_1(\overline{y})\}$ and $\{\overline{g}(\overline{z}) : P_2(\overline{z})\}$ and $h = \phi_{P_1,P_2}(f,g)$ and $prec(\phi,P_1,P_2,f,g)$. From $R_1^L(P_1(\overline{y}))$ and $R_2^L(P_2(\overline{z}))$ it follows that $R_1(\overline{f}(\overline{y}))$ and $R_2(\overline{g}(\overline{z}))$. From $\forall\overline{f}\overline{g}(R_1(\overline{f}(\overline{y}))\wedge R_2(\overline{g}(\overline{z})) \supset R(\phi_{P_1,P_2}(\overline{f},\overline{g})(\overline{y}\circ\overline{z})))$ it follows that $R(\phi_{P_2,P_2}(\overline{f},\overline{g})(\overline{y}\circ\overline{z})))$ Therefore $R(\overline{h}(\overline{y}\circ\overline{z})))$, so $R(\overline{y}'\circ\overline{z}')$. Therefore $\{\overline{y}'\circ\overline{z}' : P_1(\overline{y});P_2(\overline{z})\} \supset R(\overline{y}',\overline{z}')$. By axiom 2, $R^L(P_1(\overline{y});P_2(\overline{z}))$.

Conditional rule Suppose $R_1^L(P_1(\overline{y}))$, $R_2^L(P_2(\overline{z}))$, $\overline{y} \subseteq FV(P_1)$, $\overline{z} \subseteq FV(P_2)$, and $\forall x'\overline{y}'\overline{z}'(x' \wedge R_1(\overline{y}')) \vee (\neg x' \wedge R_2(\overline{z}')) \supset R(x',\overline{y}',\overline{z}')$. Suppose also that $\{x',\overline{y}',\overline{z}' : x?P_1(\overline{y})?P_2(\overline{z})\}$. By the conditional axiom 37, $(x' = \mathbf{true} \wedge \{\overline{y}' : [\overline{y}]P_1(\overline{y})\}) \vee (x' = \mathbf{false} \wedge \{\overline{z}' : [\overline{z}]P_2(\overline{z})\})$. From $R_1^L(P_1(\overline{y}))$ and $R_2^L(P_2(\overline{z}))$ it follows that $(x' = \mathbf{true} \wedge R_1(\overline{y}')) \vee (x' = \mathbf{false} \wedge R_2(\overline{z}'))$. From $\forall x'\overline{y}'\overline{z}'(x' \wedge R_1(\overline{y}')) \vee (\neg x' \wedge R_2(\overline{z}')) \supset R(x',\overline{y}',\overline{z}')$. it follows that $R(x',\overline{y}',\overline{z}')$. Therefore $\{x',\overline{y}',\overline{z}' : x?P_1(\overline{y})?P_2(\overline{z})\} \supset R(x',\overline{y}',\overline{z}')$. By axiom 2, $R^L([x,\overline{y},\overline{z}]x?P_1(\overline{y})?P_2(\overline{z}))$.

Output deletion rule Suppose $R^L([\overline{u}, x, \overline{v}]P)$ and $R_1(\overline{u}, \overline{v}) \equiv \exists x R(\overline{u}, x, \overline{v})$. Suppose $\{\overline{u}', \overline{v}' : [\overline{u}, \overline{v}]\exists x P\}$. By the deleting output axiom,

$$\forall \overline{u}'\overline{v}'(\exists x'\{\overline{u}', x', \overline{v}' : [\overline{u}, x, \overline{v}]P\} \equiv \{\overline{u}', \overline{v}' : [\overline{u}, \overline{v}]\exists x P\}) \tag{56}$$

Therefore $\exists x'\{\overline{u}', x', \overline{v}' : [\overline{u}, x, \overline{v}]P\}$. Because $R^L([\overline{u}, x, \overline{v}]P)$, it follows that $\exists x' R(\overline{u}', x', \overline{v}')$. Therefore by definition of R_1, $R_1(\overline{u}', \overline{v}')$. Because $\{\overline{u}', \overline{v}' : [\overline{u}, \overline{v}]\exists x P\}$ implies $R_1(\overline{u}', \overline{v}')$, therefore $R_1^L([\overline{u}, \overline{v}]\exists x P)$.

Procedure rule Suppose $R^L(P(\overline{x}, \overline{y}))$ and $R_1(\overline{u}, q) \equiv \forall \overline{v}(q(\overline{v}) \supset R(\overline{u}, \overline{v}))$. It is necessary to show $\forall \overline{u}\exists q R_1(\overline{u}, q)$ and $R_1^L([\overline{x}, p] \uparrow_{\overline{y}}^{p} P(\overline{x}, \overline{y}))$. By the procedure operator axioms,

$$\forall \overline{u}\exists q\{\overline{u}, q : [\overline{x}, p] \uparrow_{\overline{y}}^{p} P(\overline{x}, \overline{y})\} \tag{57}$$

and

$$\forall pq\overline{u}(\{\overline{u}, q : [\overline{x}, p] \uparrow_{\overline{y}}^{p} P(\overline{x}, \overline{y})\} \supset \forall \overline{v}(\{\overline{u}, \overline{v} : [\overline{x}, \overline{y}]P(\overline{x}, \overline{y})\} \equiv q(\overline{v}))) \tag{58}$$

where p is a new variable. From equations 57 and 58 it follows that

$$\forall \overline{u}\exists q\forall \overline{v}(\{\overline{u}, \overline{v} : [\overline{x}, \overline{y}]P(\overline{x}, \overline{y})\} \equiv q(\overline{v})). \tag{59}$$

By the definition of $R^L(P(\overline{x}, \overline{y}))$, it follows that $\{\overline{u}, \overline{v} : [\overline{x}, \overline{y}]P(\overline{x}, \overline{y})\} \supset R(\overline{u}, \overline{v})$. From this and equation 59 it follows that $\forall \overline{u}\exists q\forall \overline{v}(q(\overline{v}) \supset R(\overline{u}, \overline{v}))$, and by the definition of R_1 this implies $\forall \overline{u}\exists q R_1(\overline{u}, q)$. From equation 58 and the fact that $R^L(P(\overline{x}, \overline{y}))$ and by the definition of R_1 it follows that

$$\forall pq\overline{u}(\{\overline{u}, q : [\overline{x}, p] \uparrow_{\overline{y}}^{p} P(\overline{x}, \overline{y})\} \supset R_1(\overline{u}, q). \tag{60}$$

Therefore $R_1^L([\overline{x}, p] \uparrow_{\overline{y}}^{p} P(\overline{x}, \overline{y}))$.

Application rule 2 Suppose $q(\overline{v}) \supset R(\overline{v}, q)$. It is necessary to show $R^L([\overline{y}, p] \downarrow_{\overline{y}}^{p})$. By the second application axiom, $\forall \overline{v}q(\{\overline{v}, q : [\overline{y}, p] \downarrow_{\overline{y}}^{p}\} \equiv q(\overline{v}))$. Because $q(\overline{v}) \supset R(\overline{v}, q)$, $\forall \overline{v}q(\{\overline{v}, q : [\overline{y}, p] \downarrow_{\overline{y}}^{p}\} \supset R(\overline{v}, q))$. Therefore $R^L([\overline{y}, p] \downarrow_{\overline{y}}^{p})$.

Least fixpoint rule Suppose

$$\forall \overline{x}'u'\Sigma\overline{y}'v'R(\overline{x}', u', \overline{y}', v'), R^L(P(\overline{x}, u, \overline{y}, v)), \overline{x}, u \in P^{in}, \overline{y}, v \in P^{out} \tag{61}$$

$$\forall \overline{x}'u'\overline{y}'v'(R(\overline{x}', u', \overline{y}', v') \supset \exists \overline{y}''v''\{\overline{x}', u', \overline{y}'', v'' : P(\overline{x}, u, \overline{y}, v)\}) \tag{62}$$

$$\forall \overline{x}'\overline{y}'w'(R_1(\overline{x}', \overline{y}', w') \equiv (R(\overline{x}', w', \overline{y}', w') \wedge \forall z'\forall \overline{y}''(R(\overline{x}', z', \overline{y}'', z') \supset z' \geq w'))) \tag{63}$$

It is necessary to show $R_1^L(\mu(u, v, P(\overline{x}, u, \overline{y}, v), \geq))$. Assume

$$\{\overline{x}', \overline{y}', w' : \mu(u, v, P, \geq)(\overline{x}, \overline{y}, v)\}. \tag{64}$$

It is necessary to show $R_1(\overline{x}', \overline{y}', w')$, that is,

$$R(\overline{x}', w', \overline{y}', w') \wedge \forall z'\forall \overline{y}''(R(\overline{x}', z', \overline{y}'', z') \supset z' \geq w'). \tag{65}$$

By the least fixpoint axiom, if

$$\forall \overline{x}' u' \Sigma \overline{y}' v' \{\overline{x}', u', \overline{y}', v' : P(\overline{x}, u, \overline{y}, v)\} \tag{66}$$

and $\overline{y}, v$ are outputs and $\overline{x}, u$ are inputs of P then

$$\begin{aligned} \{\overline{x}', \overline{y}', w' : \mu(u, v, P, \geq)(\overline{x}, \overline{y}, v)\} \quad &\equiv \\ (\{\overline{x}', w', \overline{y}', w' : P(\overline{x}, v, \overline{y}, v)\} \quad &\wedge \quad \forall z' \forall \overline{y}''(\{\overline{x}', z', \overline{y}'', z' : P(\overline{x}, u, \overline{y}, v)\} \supset z' \geq w')). \end{aligned} \tag{67}$$

Now, formula 66 follows from formula 61. For if $\{\overline{x}', u', \overline{y}', v' : P(\overline{x}, u, \overline{y}, v)\}$ and $\{\overline{x}', u', \overline{y}'', v'' : P(\overline{x}, u, \overline{y}, v)\}$ then by formula 61, $R^L(P(\overline{x}, u, \overline{y}, v))$, so $R(\overline{x}', u', \overline{y}', v')$ and $R(\overline{x}', u', \overline{y}'', v'')$ hold, and from the formula $\forall \overline{x}' u' \Sigma \overline{y}' v' R(\overline{x}', u', \overline{y}', v')$ it follows that $\overline{y}' = \overline{y}''$ and $v' = v''$. This proves formula 66. Then by the least fixpoint axiom, formula 67 follows. From formula 67 and assumption 64 it follows that $\{\overline{x}', w', \overline{y}', w' : P(\overline{x}, v, \overline{y}, v)\}$, and therefore from $R^L(P(\overline{x}, u, \overline{y}, v))$ it follows that $R(\overline{x}', w', \overline{y}', w')$. To prove formula 65, it is also necessary to show

$$\forall z' \forall \overline{y}''(R(\overline{x}', z', \overline{y}'', z') \supset z' \geq w'). \tag{68}$$

Suppose $R(\overline{x}', z', \overline{y}'', z')$. From formula 62 it follows that formula $\{\overline{x}', z', \overline{y}''', z'' : P(\overline{x}, u, \overline{y}, v)\}$ holds for *some* $\overline{y}'''$ and z'', and therefore $R(\overline{x}', z', \overline{y}''', z'')$. Since $\overline{y}'''$ and z'' are unique by formula 61, they are equal to $\overline{y}''$ and z'. Therefore $\{\overline{x}', z', \overline{y}'', z' : P(\overline{x}, u, \overline{y}, v)\}$, and then from formula 67 it follows that $z' \geq w'$. This completes the proof.

There is an algorithm to extract L programs from proofs in relational $PL(L)$, as follows:

Definition 5.1 *The PL* program operations *are composition* $(;)$, *conditional* $(?)$, *variable deletion* $(\exists)$, *procedure* $(\uparrow)$, *application* $(\downarrow)$, *and least fixpoint* (μ).

Definition 5.2 *If L is a PL-feasible language, and $P_1, P_2, \ldots, P_n$ are L programs, then an L* program term *over $P_1, P_2, \ldots, P_n$ is either*

1. *one of the programs P_i, or*
2. *of the form $P;Q$, $x?P?Q$, $\uparrow^p_{\overline{x}} P$, $\downarrow^p_{\overline{x}} P$, $\exists x P$, or $fp(x, y, P, \geq)$ where P and Q are L program terms over $P_1, P_2, \ldots, P_n$ and x, y, p, and $\overline{x}$ are program variables, and where the preconditions for these operators are satisfied.*

Theorem 4 *If L is a PL-feasible language and there is a proof of an assertion of the form $R^L([\overline{y}]P)$ from assertions of the form $R^L_i([\overline{y}^i]P_i)$ in relational $PL(L)$, then P is expressible as an L program term over $P_1\Theta_1$, $P_2\Theta_2$, ..., $P_n\Theta_n$ for some output injective variable substitutions Θ_i.*

Proof: By induction on proof depth. For depth 0, $P = P_i$ for some i, and P_i is trivially an L program term over $P_1, \ldots, P_n$. Assume the theorem is true for proofs of depth d. A proof of depth $d+1$ consists of one or two proofs of depth d followed by the application of an inference rule. By induction, the theorem is true for the proof or proofs of depth d. Then, using the forms of the inference rules, the theorem is also true for the proof of depth $d+1$.

It is necessary to look at each inference rule. For the underlying logic rule, P is not altered, so the induction step holds. For permutation rules 1 and 2, only the variables of P are renamed, and renaming variables in an L program term over $P_1\Theta_1, \ldots, P_n\Theta_n$ yields another L program term over $P_1\Theta_1', \ldots, P_n\Theta_n'$ for suitable Θ_i'. The equality rule does not alter P. The substitution rule applies an integer function σ to the variables of P. This can be incorporated into the Θ_i as well, but it is necessary to check that identifying variables of the P_i does not invalidate any inference rules used to obtain P. The correspondence rule does not affect P, only its preceding list of variables. The definitional independence rule is similar to the substitution rule in its effect on P. The remaining rules (composition rule, conditional rule, output deletion rule, procedure rule, application rule, and least fixpoint rule) all produce L program terms from L program terms.

Now, the PL program operations of composition, conditional, output deletion, $\uparrow$, $\downarrow$, and μ have some preconditions, and it is necessary to check that these preconditions still hold in the resulting L program term after applying the substitution rule and the definitional independence rule. The sequential composition operator $P;Q$ requires that no variable be in $P^{out} \cap Q^{out}$ and that no data variable be in $P^{in} \cap Q^{out}$. Assuming the former condition is true when the ; operator is applied, it will remain true because all substitutions are output injective. The latter condition on data variables will remain true because no substitution identifies two data variables unless both are inputs. The procedure operator axiom for $\uparrow^p P$ requires that p be a new variable or an input variable for P. Now, p is an output variable of $\uparrow^p P$, which implies that no substitution will identify p with any other output variable (because substitutions are output injective). Thus any substitution Θ will only identify P with input variables, so p will still be an input variable in $P\Theta$ and the preconditions for this rule will still hold. The preconditions for the application axiom are similar to those for composition, and similar reasoning applies. The preconditions for $\mu(u, v, P, \geq)$ state that u is an input and v is an output to P. Also, u is not a free variable of $\mu(u, v, P, \geq)$ and v is an output. Because of the rules for applying substitutions, u and v will remain distinct, and u will remain an input variable after substitutions are applied. v will remain an output variable as well, because the common image of an input and an output variable is an output variable, so the preconditions for μ will continue to hold.

Corollary 1 *If in addition the PL program operations composition, conditional, $\exists$, $\uparrow$, $\downarrow$, and μ are effectively computable in L, in the sense that an L program for $P;Q$ can effectively be obtained from L programs for P and Q, et cetera, then an L program P such that $R^L([\overline{y}]P)$ is effectively computable from the proof, given L programs for P_1, P_2, ..., P_n.*

6 Abstract Inference Rules

The preceding inference rules permit proofs of properties of programs in a specific programming language L. It is possible to modify these rules to obtain the system PL^* that permits abstract proofs of the existence of programs, but not in a specific language. Such proofs can then can be translated into programs in specific PL-feasible programming languages automatically. These abstract rules involve assertions of the form $(\exists P)R^L(P)$ where P is a variable representing a program and R is a relation on programs and L is a variable representing a PL-feasible language. The form of the proof not only guarantees that such a program P exists, but also permits a specific program to be derived from the proof, as in other program generation systems. It is necessary to record the list of input and output variables for each program variable P in order to use these inference rules; rules appearing in section 5 suffice to compute these lists for variables P appearing in the conclusion of each rule.

$$\frac{(\exists P)R_1^L(P), R_1 \supset R_2}{(\exists P)R_2^L(P)} \quad \text{Abstract underlying logic rule}$$

$$\frac{(\exists P)R_k^L(P), k \text{ arbitrary}}{(\exists P)\forall k R_k^L(P)} \quad \text{Abstract universal quantification rule}$$

Rules about variables

$$\frac{(\exists P)R^L([\overline{x}]P(\overline{x})), \Theta \text{ is a variable renaming}}{(\exists P)R^L([\overline{x}\Theta]P(\overline{x}\Theta))} \quad \text{Abstract variable renaming rule}$$

$$\frac{(\exists P)R^{L,\overline{\psi}}([\overline{x}]P(\overline{x})), \sigma \text{ is an integer permutation}}{(\exists P)R\sigma^{L,\overline{\psi}}([\overline{\sigma}^{-1}(\overline{x})]P(\overline{x}))} \quad \text{Abstract permutation rule 1}$$

$$\frac{(\exists P)R^L([\overline{x}]P(\overline{x})), \sigma \text{ is an integer permutation}}{(\exists P)R\sigma^L([\overline{x}]P(\overline{\sigma}(\overline{x})))} \quad \text{Abstract permutation rule 2}$$

$$\frac{(\exists P)R^L(P(\overline{x})), x_i \equiv x_j, R_1(\overline{y}) \equiv (R(\overline{y}) \wedge y_i = y_j)}{(\exists P)R_1^L(P(\overline{x}))} \quad \text{Abstract equality rule}$$

$$\frac{\begin{array}{c}(\exists P)R^L([\overline{x}]P(\overline{x})), \sigma \text{ is an integer function such that} \\ \overline{\sigma} \text{ does not identify data variables of } \overline{x}\end{array}}{(\exists P)R\sigma^L([\overline{x}]P(\overline{\sigma}(\overline{x})))} \quad \text{Abstract substitution rule}$$

$$\frac{(\exists P)R^L([\overline{x}]P), \forall f, g(R(\overline{f}(\overline{x})) \wedge \forall u \in FV(P) \cup ([\overline{x}]P)^{data}(f(u) = g(u))) \supset R_1(\overline{g}(\overline{y}))}{(\exists P)R_1^L([\overline{y}]P)} \quad \begin{array}{c}\text{Abstract} \\ \text{correspondence rule}\end{array}$$

$$\frac{\begin{array}{c}(\exists P)R^L([\overline{x}]P(\overline{x})), \Theta \text{ output injective on } P \\ \text{and does not identify data variables of } P\end{array}}{(\exists P)R^L([\overline{x}\Theta]P(\overline{x}\Theta))} \quad \text{Abstract definitional independence rule}$$

$$\frac{(\exists P)R^{L,\psi}([\overline{x}, \psi]P), y \text{ a new data variable}, R(\overline{u}, v) \wedge R(\overline{u}, w) \supset R_1(\overline{u}, v, w)}{(\exists P)R_1^{L,\psi}([\overline{x}, y, \psi]P)} \quad \text{Abstract new variable rule}$$

Rules about program operations

$$\frac{\begin{array}{c}(\exists P_1)R_1^L(P_1(\overline{y})),(\exists P_2)R_2^L(P_2(\overline{z})),\\ prec(\phi,P,Q,f,g),\forall\overline{f}\overline{g}(R_1(\overline{f}(\overline{y}))\wedge R_2(\overline{g}(\overline{z}))\supset R(\phi_{P,Q}(\overline{f},\overline{g})(\overline{y}\circ\overline{z})))\end{array}}{(\exists P)R^L(P(\overline{y},\overline{z}))}\quad \text{Abstract composition rule}$$

$$\frac{\begin{array}{c}(\exists P_1)R_1^L(P_1(\overline{y})),(\exists P_2)R_2^L(P_2(\overline{z})),\overline{y}\subseteq FV(P_1),\overline{z}\subseteq FV(P_2),\\ \forall x'\overline{y}'\overline{z}'(x'\wedge R_1(\overline{y}'))\vee(\neg x'\wedge R_2(\overline{z}'))\supset R(x',\overline{y}',\overline{z}')\end{array}}{(\exists P)R^L(P(x,\overline{y},\overline{z}))}\quad \text{Abstract conditional rule}$$

$$\frac{(\exists P)R^L([\overline{u},x,\overline{v}]P),R_1(\overline{u},\overline{v})\equiv\exists xR(\overline{u},x,\overline{v})}{(\exists P)R_1^L(P)}\quad \text{Abstract output deletion rule}$$

$$\frac{(\exists P)R^L(P(\overline{x},\overline{y})),R_1(\overline{x}',q)\equiv\forall\overline{y}'(q(\overline{y}')\supset R(\overline{x}',\overline{y}'))}{\forall\overline{x}'\exists qR_1(\overline{x}',q)\wedge(\exists P)R_1^L(P)}\quad \text{Abstract procedure rule}$$

$$\frac{q(\overline{v})\supset R(\overline{v},q)}{(\exists P)R^L([\overline{y},p]P)}\quad \text{Abstract application rule}$$

$$\frac{\begin{array}{c}\forall\overline{x}'u'\Sigma\overline{y}'v'R(\overline{x}',u',\overline{y}',v'),\\ (\exists P)(R^L(P(\overline{x},u,\overline{y},v)),\overline{x},u\in P^{in},\overline{y},v\in P^{out},\\ \wedge\forall\overline{x}'u'\overline{y}'v'(R(\overline{x}',u',\overline{y}',v')\supset\exists\overline{y}''v''\{\overline{x}',u',\overline{y}'',v'':P(\overline{x},u,\overline{y},v)\})),\\ \forall\overline{x}'w'\overline{y}'(R_1(\overline{x}',\overline{y}',w')\equiv(R(\overline{x}',w',\overline{y}',w')\wedge\forall z(R(\overline{x}',z',\overline{y}',z')\supset z'\geq_L w')))\end{array}}{(\exists P)R_1^L(P)}\quad \begin{array}{c}\text{Abstract least}\\ \text{fixpoint rule}\end{array}$$

It is possible to translate proofs in PL^* into programs in *any* PL-feasible language M:

Theorem 5 *There is an algorithm which, given a PL^* proof of an assertion of the form $\exists PR^L([\overline{y}]P)$ from assertions of the form $\exists P_iR_i^L([\overline{y}^i]P_i)$ (where L is a variable representing a PL-feasible language), and given a PL-feasible language M and M programs P_i such that $R^M([\overline{y}^i]P_i)$, produces an M-program P such that $R^M([\overline{y}]P)$.*

Proof: It is straightforward to translate PL^* proofs into $PL(L)$ proofs, for any PL-feasible language L, and then apply the algorithm of corollary 1.

7 Abstract programs

Corresponding to abstract inference rules there are *abstract programs* in PL.

Definition 7.1 *An* abstract PL program *is either*

1. *A variable X, representing a program fragment, or*

2. *Of the form $P; Q$, $x?P?Q$, $\uparrow_{\overline{x}}^{p} P$, $\downarrow_{\overline{x}}^{p} P$, $\exists x P$, or $fp(x, y, P, \geq)$ where P and Q are abstract PL programs and x, y, p, and $\overline{x}$ are program variables.*

L^* is the set of abstract PL programs. The notation $P[X_1, X_2, \ldots, X_n]$ refers to an abstract PL program, where P is a composition of the PL program operations $;, ?, \uparrow, \downarrow, \exists$, and fp and $X_1, \ldots, X_n$ is a listing of all the variables in P representing program fragments. By contrast, $P(\overline{x})$ represents the program P mentioning the program variables $\overline{x}$. If $P_1, P_2, \ldots, P_n$ are L programs for some PL-feasible language L, then $P[P_1, P_2, \ldots, P_n]$ denotes the L program term that results from replacing all occurrences of X_i in $P[X_1, \ldots, X_n]$ by P_i. The program variables $\overline{x}$ can also be indicated as in $P[P_1(\overline{x}), P_2(\overline{x}), \ldots, P_n(\overline{x})]$.

By theorem 5, a PL^* proof can be converted to a $PL(L)$ proof, for any PL-feasible L, and from the $PL(L)$ proof an L program term can be obtained. This L program term is much like an abstract PL program, but it contains substitutions on the programs P_i. It is possible to eliminate these substitutions and also the dependence on the particular proof system $PL(L)$ or PL^*, as follows.

Definition 7.2 *If $P(\overline{x})$ is a $PL(L)$-program having $\overline{x}$ as free variables, then $[[P(\overline{x})]]$, the* semantics *of $P(\overline{x})$, is the set $\{f : \{\overline{f}(\overline{x}) : P(\overline{x})\}\}$, where f is a function from variables to their semantics.*

Theorem 6 *If P and Q are L programs, then $[[P; Q]]$ is a function of $[[P]]$ and $[[Q]]$, $[[\exists x P]]$ is a function of $[[P]]$, $[[\uparrow_{\overline{x}}^{p} P]]$ is a function of $[[P]]$, and similarly for the other PL program operations.*

Proof: By consideration of the definition of each operation, noting that the semantics of the operations depend only on the semantics of the operands.

Definition 7.3 *Extend the PL operations on programs to operations on their semantics, so that $[[P; Q]] = [[P]]; [[Q]]$, $[[\exists x P]] = \exists x [[P]]$, and so on, thus giving names to the functions in theorem 6*

Theorem 7 *For every L program $P[P_1, \ldots, P_n]$ where P is an abstract PL program (composed of the PL program operations) and P_i are variables representing L programs, there is a function f_P^L depending on P but not on $P_1, \ldots, P_n$ such that*

$$f_P^L([[P_1]], \ldots, [[P_n]]) = [[P[P_1, \ldots, P_n]]] \tag{69}$$

for all L programs $P_1, \ldots, P_n$.

Proof: By induction on the depth of P, using Theorem 6.

Theorem 8 *For every abstract L program $P[X_1, \ldots, X_n]$ where P is an abstract PL program (composed of the PL program operations) and X_i are variables representing program fragments, there is a function f_P^* depending on P but not on $X_1, \ldots, X_n$ such that*

$$f_P^*([[X_1]], \ldots, [[X_n]]) = [[P[X_1, \ldots, X_n]]] \tag{70}$$

for all $X_1, \ldots, X_n$.

Proof: By induction on the depth of P, using Theorem 6 and Definition 7.3

Theorem 9 *For any PL feasible language L and any L program $P[P_1, \ldots, P_n]$ where P is an abstract PL program, the abstract program $P[X_1, \ldots, X_n]$ satisfies $f_P^L = f_P^*$.*

Proof: $f_P^L([[P_1]], \ldots, [[P_n]]) = [[P[P_1, \ldots, P_n]]] = f_P^*([[P_1]], \ldots, [[P_n]])$ by Theorems 7 and 8.

Corollary 2 *Suppose that $P[X_1, \ldots, X_n]$ is an abstract PL program and $A_1, \ldots, A_n, A$ are assertions such that for all program fragments $X_1, \ldots, X_n$, $A_1([[X_1]]) \wedge \ldots \wedge A_n([[X_n]]) \supset A([[P[X_1, \ldots, X_n]]])$. Then for any PL feasible language L and any L programs $P_1, \ldots, P_n$ of appropriate sorts, $A_1([[P_1]]) \wedge \ldots \wedge A_n([[P_n]]) \supset A([[P(P_1, \ldots, P_n)]])$.*

Proof: From the hypothesis $A_1([[X_1]]) \wedge \ldots \wedge A_n([[X_n]]) \supset A([[P[X_1, \ldots, X_n]]])$ and Theorem 8 it follows that $A_1([[X_1]]) \wedge \ldots \wedge A_n([[X_n]]) \supset A(f_P^*([[X_1]], \ldots, [[X_n]]))$. From Theorem 9 it follows that $A_1([[X_1]]) \wedge \ldots \wedge A_n([[X_n]]) \supset A(f_P^L([[X_1]], \ldots, [[X_n]]))$. From Theorem 7 it follows that $A_1([[P_1]]) \wedge \ldots \wedge A_n([[P_n]]) \supset A([[P[P_1, \ldots, P_n]]])$.

This yields the following method for constructing L programs satisfying a specification:

1. Construct an abstract PL program $P[X_1, \ldots, X_n]$.
2. Show that P satisfies the specification $A_1([[X_1]]) \wedge \ldots \wedge A_n([[X_n]]) \supset A([[P[X_1, \ldots, X_n]]])$.
3. Choose L programs $P_1, \ldots, P_n$.
4. Show that these programs satisfy $A_1([[P_1]]), \ldots, A_n([[P_n]])$.
5. Conclude that the L program $P[P_1, \ldots, P_n]$ satisfies the specification $A([[P[P_1, \ldots, P_n]]])$.

This shows that one can construct abstract PL programs satisfying a specification, and from them one can construct L programs satisfying the specification, for any PL-feasible language L. L^* programs are somewhat similar to "pseudocode" descriptions of algorithms found in textbooks, but unlike pseudocode, L^* programs have a formal syntax and semantics, which permit programs to be verified. It would of course be possible to verify a program P in some particular language such as C and translate P to other languages L. Why is L^* any better for this purpose? The syntax and semantics of L^* are simple, making it easier to write such a translator and the translator is more likely to produce efficient code in L. It is also easier to verify L^* programs than C programs.

Another possibility would be to verify a program in lambda calculs or μ calculus or some other language with a simple syntax and semantics and translate this program into other languages. An advantage of L^* is that it has features to guide the translation, such as the distinction between procedures and data, the use of ; to signify sequential composition, the use of $\exists$ to signify variable declarations, the use of conditionals, and so on. This means that in the abstract program one can give guidance about how the algorithm should be expressed to gain efficiency.

In fact, an abstract program can be considered as a way to formally describe algorithms. A description of an algorithm in a particular programming language gives extraneous details related to the programming language syntax but not to the algorithm. A pseudocode description of algorithms as found in textbooks does not have a precise syntax and semantics. Turing machine descriptions also contain extraneous details and lack abstraction and do not capture the efficiency of data structures. Pure functional languages without destructive assignments do not permit an imperative programming style, which can lead to inefficiency. More abstract notations such as lambda calculus and μ calculus give too little guidance concerning efficient code generation, which generally requires destructive modification of data structures, side effects, and conditional statements, and are difficult to translate efficiently into more conventional programming languages. Thus PL is abstract enough to avoid extraneous details about syntax but not too abstract to express program features that have a major influence on efficiency.

The emphasis of PL is not so much the automatic construction of programs or even automatic proofs of their correctness, but rather the ability to write abstract proofs or programs that can be translated into a wide variety of other languages, to avoid the necessity of writing the same program over and over again in different languages. Probably it would be most efficient for the abstract programs to be coded by humans and stored in a library. It does not appear feasible to construct complex programs by automatic program generation methods in most cases. The PL approach permits a reduction of programmer effort even in the absence of automatic program generation. The system PL can even be used without formal proofs of correctness; the programs P_i can be verified to satisfy the assertions A_i, or this can just be checked by testing, to gain some measure of reliability without a formal proof. In fact, it is not even necessary to know that the language L is PL-feasible; this can be verified in a large number of cases by testing, to gain some confidence in the reliability of the programs.

Abstract programs may be parameterized. For example, the precision of floating point operations may be a parameter. If this precision is too high, then the abstract program may not translate into as many languages. Another example of a parameter might be the length of character strings. If different languages implement different length character strings, then depending on the values of this parameter, the abstract program would translate into a different set of languages. However, if the abstract program is correct regardless of the parameter values, then the L programs resulting from it will also be correct for all values of the parameters.

The abstract programs are not necessarily easy to read or understand, although readability is easier for the textual syntax. Here is an abstract program for factorial:

$$\mu(f, g, \uparrow^{g}_{n,v} (\exists twxyz \downarrow^{0}_{t}; \downarrow^{=}_{ntw}; w? \downarrow^{1}_{v}? \downarrow^{1}_{x}; \downarrow^{-}_{nxy}; \downarrow^{f}_{yz}; \downarrow^{*}_{nzv}), \geq)$$

Another approach is to allow the defined symbol g to be one (f) that already appears, yielding the following simpler program:

$$\uparrow^{f}_{n,v} (\exists twxyz \downarrow^{0}_{t}; \downarrow^{=}_{ntw}; w? \downarrow^{1}_{v}? \downarrow^{1}_{x}; \downarrow^{-}_{nxy}; \downarrow^{f}_{yz}; \downarrow^{*}_{nzv})$$

This program has the following PL textual syntax:

proc $f(n, v)$;
  var t, w, x, y, z;
  call $0(t)$;
  call $=(n, t, w)$;
  if w **then call** $1(v)$ **else call** $1(x)$; **call** $-(n, x, y)$; **call** $f(y, z)$; **call** $*(n, z, v)$ **fi**
end f;

Here $f(n, v)$ is a procedure with input n and output v, $0(t)$ sets t to zero, $= (n, t, w)$ sets w to **true** if $n = t$, **false** otherwise, $1(v)$ sets v to one, $-(n, x, y)$ sets y to $n - x$, and $*(n, z, v)$ sets v to $n * z$. If w is true then v is set to 1 else x is set to 1, y is set to $n - x$, $f(y, z)$ is called, and v is set to $n * z$. The μ operator is not necessary in this case. In fact, for many PL feasible L, it is never necessary to use μ, because $\mu(u, v, P[u, v, \overline{x}], \geq) = P[v, v, \overline{x}]$.

The abstract programs could be made more abstract in various ways, such as making them polymorphic.

As an example of the use of data, consider the following program to update all elements of an array of length n:

proc Update(A,n)
  call Update1(A,n);
  if $n > 1$ **then call Update**(A,$n-1$) **fi**;
end Update;

Here **Update1**(A,n) returns array A^{fin} with the n^{th} element updated. The variables A and n are data variables and **Update** and **Update1** are procedures. Without the use of data variables, one would have to recopy the whole array to update each element. The fact that the parameter A to **Update1** is both an input and an output of the procedure avoids this inefficiency.

In A^* notation, without syntactic sugar, the above program would be

$$\uparrow^{Update}_{An} (\downarrow^{Update1}_{An}; \exists xyz(\downarrow^{1}_{x}; \downarrow^{-}_{nxy}; \downarrow^{=}_{y0z}; z? \downarrow^{Update}_{Ay} ?))$$

One can obtain the effect of global variables as data variables that are inputs to several procedures, as follows:

$$\uparrow^{p}_{\overline{x}} P; \uparrow^{q}_{\overline{y}} Q$$

where $u \in P^{in} \cap Q^{in}$ and u is a data variable. Input and output are essentially global data variables representing files, and read and write statements modify these variables.

One can obtain iterative loops by recursion, or as special procedures that are known to the compiler and that permit compilation by iteration instead of recursion. They can also be added as operators to PL, much as the conditional operator was added.

8 An example: Quicksort

This section presents an example program, quicksort, as an abstract program with an associated proof of correctness. A sketch of a translation of the abstract program into a quicksort program in C follows. This example illustrates the proof rules as well as the importance of destructive modification of data for program efficiency. It is not necessary to consider side effects because no operations in the quicksort program have side effects.

The textual syntax for the Quicksort program, with some simplifications, is the following:

```
proc Quicksort(A,p,r)
  var q;
  if p < r
  then Partition(A, p, r, q);
    Quicksort(A, p, q);
    Quicksort(A, q + 1, r)
  fi
end Quicksort
```

This corresponds to the abstract program

$$\uparrow^{Q}_{Apr} \exists qxy(\downarrow^{lt}_{prx}; x?(\downarrow^{P}_{Aprq}; \downarrow^{Q}_{Apq}; \downarrow^{inc}_{qy}; \downarrow^{Q}_{Ayr}))$$

where Q is **Quicksort**, P is **Partition**, $lt(p, r, x)$ sets x to **true** if $p < r$, **false** otherwise, and $inc(q, y)$ sets y to $q+1$. Define abstract programs P_1, P_2, P_3, P_4, P_5, and P_6, respectively, as follows:

$$P_1(Q, A, p, q) = \downarrow^{Q}_{Apq}$$

$$P_2(P, A, p, q, r) = \downarrow^{P}_{Aprq}$$

$$P_3(P, Q, A, p, q, r) = P_2(P, A, p, q, r); P_1(Q, A, p, q)$$

$$P_4(P, Q, A, p, q, r, y) = P_3(P, Q, A, p, q, r); \downarrow^{inc}_{qy}; \downarrow^{Q}_{Ayr}$$

$$P_5(P, Q, A, p, q, r, x, y) = \downarrow^{lt}_{prx}; x?P_4(P, Q, A, p, q, r, y)$$

$$P_6(P, Q) = \uparrow^{Q}_{Apr} \exists qxy P_5(P, Q, A, p, q, r, x, y)$$

If L is the language C, then $P_1(Q, A, p, q)$ might correspond to the statement "$Q(A, p, q)$;", $P_2(P, A, p, q, r)$ might correspond to the statement "$P(A, p, q, r)$;", $P_3(P, Q, A, p, q, r)$ might correspond to the sequence "$P(A, p, q, r); Q(A, p, q)$;" of two procedure calls, et cetera. The program fragment $\exists qxy$ would correspond to the declarations "int q, x, y;" in this case, assuming q, x, and y have integer sorts, but could correspond to the statement "float q, x, y;" if q, x, and y had real number (floating point) sorts. The C program for $P_6(P, Q)$ would be something like "void $Q(A, p, r)$ int $A[], p, r$; int q, x, y; { $P_5^L(P, Q, A, p, q, r, x, y)$ }" where L is C. The C program for $P_6(P, R)$ would be "void $R(A, p, r)$ int $A[], p, r$; int q, x, y; { $P_5^L(P, R, A, p, q, r, x, y)$ }", showing how program variables (names of procedures or data variables) in program text can be replaced. Quicksort programs in other languages besides C could be generated in a similar manner.

To give a proof of correctness, define $perm(A, B)$ for two arrays A and B to specify that the elements of B are a permutation of the elements of A, and define relations as follows:

$$R_{perm}(A) \equiv perm(A^{init}, A^{fin})$$

$$R_{bdry}(x, y) \equiv (x^{init} = x^{fin} \wedge y^{init} = y^{fin})$$

$$R_{split}(A, p, q, r) \equiv \forall ij((p^{init} \leq i \leq q^{fin} \wedge q^{fin} < j \leq r^{init}) \supset A^{fin}[i] \leq A^{fin}[j])$$

$$R_{part}(A, p, q, r) \equiv R_{perm}(A) \wedge R_{bdry}(p, r) \wedge (p^{init} \leq q^{fin}) \wedge (q^{fin} < r^{init}) \wedge R_{split}(A, p, q, r)$$

$$R_{sort}(A,x,y) \equiv \forall ij(x \le i < j \le y \supset A^{fin}[i] \le A^{fin}[j])$$

For convenience, it helps to identify program variables with their semantics when defining relations. $R^L(P(\overline{x}))$ is defined to mean $\forall\overline{y}(\{\overline{y} : P(\overline{x})\} \supset R(\overline{y}))$. This can also be written as $f \in [[P(\overline{x})]] \supset R(f(x_1)\ldots f(x_n))$. Defining $R_f(x_1,\ldots,x_n)$ as $R(f(x_1),\ldots,f(x_n))$, one has $f \in [[P(\overline{x})]] \supset R_f(\overline{x})$. It is more convenient to give the relations R_f than R because R_f mentions the program variables $\overline{x}$ rather than their semantics $\overline{y}$. It is these relations R_f rather than R that follow. For example, in order to express that $\forall y_1y_2(\{y_1,y_2 : P(x_1,x_2)\} \supset y_2 = y_1+1)$, one would ordinarily define $R(y_1,y_2) \equiv (y_2 = y_1+1)$, but identifying program variables x_1,x_2 with their semantics y_1,y_2 one specifies $R(x_1,x_2) \equiv (x_2 = x_1+1)$, which is more intuitive because x_1 and x_2 appear in P.

Recall that data variable x in a program fragment P has semantics (α,β) where α is the initial value of x and β is the final value. By convention, $(\alpha,\beta)^{init} = \alpha$ and $(\alpha,\beta)^{fin} = \beta$. If one identifies variables with their semantics, $\alpha = x^{init}$ and $\beta = x^{fin}$.

Define relations R_1,R_2,R_3,R_4, and R_5 to be satisfied by the programs P_1,P_2,P_3,P_4, and P_5, respectively, and relation R^k_{qs} as follows:

$$R_1(A,p,q) \equiv R_{perm}(A) \wedge R_{bdry}(p,q) \wedge R_{sort}(A,p^{init},q^{init})$$
$$R_2(A,p,q,r) \equiv ((p<r) \supset R_{part}(A,p,q,r))$$
$$R_3(A,p,q,r) \equiv ((p<r) \supset R_{part}(A,p,q,r) \wedge R_{sort}(A,p^{init},q^{fin}))$$
$$R_4(A,p,q,r) \equiv ((p<r) \supset R_1(A,p,r))$$
$$R_5(A,p,q,r) \equiv R_1(A,p,r)$$
$$R^k_{qs}(Q) \equiv \forall A'p'q'((q'-p') \le k \wedge Q(A',p',q') \supset R_1(A',p',q'))$$
$$R_{qs}(Q) \equiv \forall A'p'q'(Q(A',p',q') \supset R_1(A',p',q'))$$

$R_{qs}(Q)$ is the final specification for the quicksort program. In order to prove correctness, it is necessary to show that for all k, $R^k_{qs}(Q)$ implies $R^{k+1}_{qs}(Q)$ and use induction. For this purpose, define relations R^k_i specifying a bound on the sizes of the modified subarrays, as follows:

$$R^k_1(A,p,q,Q) \equiv (R^k_{qs}(Q) \wedge (q-p) \le k \supset R_1(A,p,q))$$
$$R^k_2(A,p,q,r) \equiv ((r-p) \le k+1 \supset R_2(A,p,q,r))$$
$$R^k_i(A,p,q,r,Q) \equiv (R^k_{qs}(Q) \wedge (r-p) \le k+1 \supset R_i(A,p,q,r)), i=3,4$$
$$R^k_5(A,p,q,r,Q) \equiv ((p \ge r) \supset A^{init} = A^{fin}) \wedge (R^k_{qs}(Q) \wedge (r-p) \le k+1 \supset R_5(A,p,q,r))$$

The formula $R^{k,L}_1([A,p,q,Q]P_1)$ follows directly from the application axiom, because $Q(A,p,q)$ holds, essentially, and because $R^k_{qs}(Q)$ holds. One then shows that if $R^L_2([A,p,q,r]P_2)$, then $R^{k,L}_3$ $([A,p,q,r,Q]P_3)$, $R^{k,L}_4([A,p,q,r,Q]P_4)$, and $R^{k,L}_5([A,p,q,r,Q]P_5)$. (For convenience the variables x, y, and P (**Partition**) are omitted.) Because k is arbitrary, one has $(\forall kR^k_5)^L([A,p,q,r,Q]P_5)$ by the universal quantification rule, and $\forall kR^k_5(A,p,q,r,Q) \equiv ((p \ge r) \supset A^{init} = A^{fin}) \wedge \forall k(R^k_{qs}(Q) \wedge (r-p) \le k+1 \supset R_5(A,p,q,r))$. From this, using the procedure rule, one derives $R^0_{qs}(Q) \wedge \forall k(R^k_{qs}(Q) \supset R^{k+1}_{qs}(Q))$ for the program $[Q]P_6$. Using the underlying logic rule and mathematical induction, $R_{qs}(Q)$ follows.

The final program does not include code for **Partition**. Any verified program for **Partition** can be inserted and will give a correct quicksort program. Thus the final verified code has some flexibility.

References

[AB00] Anderson and Basin. Program development schemata as derived rules. *Journal of Symbolic Computation*, 30(1):5–36, 2000.

[AB01] Abdelwaheb Ayari and David Basin. A higher-order interpretation of deductive tableau. *Journal of Symbolic Computation*, 31(5):487–520, 2001.

[AFM99] A. Avellone, M. Ferrari, and P. Miglioli. Synthesis of programs in abstract data types. In P. Flener, editor, *Proceedings of LOPSTR'98*, pages 81–100, 1999.

[BC85] J.L. Bates and R.L. Constable. Proofs as programs. *Transactions on Programming Languages and Systems*, 7(1):113–136, 1985.

[Ben00] V. Benini. Representing object code. In *Computational Logic - CL 2000*, pages 538–552, 2000. LNCS volume 1861.

[BF97] H. Büyükyildiz and P. Flener. Generalised logic program transformation schemas. In N.E. Fuchs, editor, *Proceedings of the Seventh International Workshop on Logic Program Synthesis and Transformation*, pages 45–65, 1997.

[BH93] Sanjay Bhansali and Mehdi T. Harandi. Synthesis of UNIX programs using derivational analogy. *Machine Learning*, 10:7–55, 1993.

[Bit92] O. Bittel. Tableau-based theorem proving and synthesis of lambda-terms in the intuitionistic logic. In *Logics in AI, volume 633 of LNCS*, pages 262–278. Springer-Verlag, 1992.

[Bor00] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[Box02] Don Box. *Essential .NET, Volume I: The Common Language Runtime*. Addison Wesley, 2002.

[BS96] U. Berger and H. Schwichtenberg. The greatest common divisor: A case study for program extraction from classical proofs. In *Lecture Notes in Computer Science Volume 1158*, pages 36–46. Springer-Verlag, Heidelberg, 1996.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Inf. and Comp*, 2/3:95–120, 1988.

[Con85] R. Constable. Constructive mathematics as a programming logic i: some principles of theory. In *Annals of Mathematics, Vol. 24*. Elsevier Science Publishers, B.V., 1985. Reprinted from Topics in the Theory of Computation, Selected Papers of the Intl. Conf. on Foundations of Computation Theory, FCT '83.

[CS93] J. N. Crossley and J. C. Shepherdson. Extracting programs from proofs by an extension of the Curry-Howard process. In J. N. Crossley, J. B. Remmel, R. Shore, and M. Sweedler, editors, *Logical Methods: Essays in honor of A. Nerode*, pages 222–288. Birkhauser, Boston, Mass., 1993.

[DL94] Yves Deville and Kung-Kiu Lau. Logic program synthesis. *Journal of Logic Programming*, 19/20:321–350, 1994.

[Dol95] A. Dold. Representing, verifying and applying software development steps using the PVS system. In *Proceedings of the Fourth International Conference on AlgebraicMethodology and Software Technology, AMAST'95*, Montreal, 1995. Springer-Verlag. volume 936 of LNCS.

[FLO98] Pierre Flener, Kung-Kiu Lau, and Mario Ornaghi. On correct program schemas. *Lecture Notes in Computer Science*, 1463:128–147, 1998.

[FLOR99] P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An abstract formalisation of correct schemas for program synthesis. *J. Symbolic Computation*, 11:1–35, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[HHZM01] M. Harman, L. Hu, X. Zhang, and M. Munro. Sideeffect removal transformation, 2001.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[JPCB00] J. Jeavons, I. Poernomo, J. N. Crossley, and B. Basit. Fred: An implementation of a layered approach to extracting programs from proofs. part I: an application in graph theory. Technical Report 2000/57, Monash University, Australia, 2000.

[Kan91] Max I. Kanovich. Efficient program synthesis: Semantics, logic, complexity. In T. Ito and editors A. Meyer, editors, *Proceedings Int'l Conf. on Theor. Aspects of Computer Software, TACS'91*, pages 24–27, Sendai, Japan, Sept. 1991. volume 526 of LNCS, pages 615–632.

[LOT99] Kung-Kiu Lau, Mario Ornaghi, and Sten-Ake Tärnlund. Steadfast logic programs. *Journal of Logic Programming*, 38(3):259–294, 1999.

[Mas97] I. A. Mason. A first-order logic of effects. *Theoretical Computer Science*, 185(2):277–318, 1997.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North-Holland.

[MW92] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, 1992.

[Par90] A. Partsch. *Specification and Transformation of Programs: a Formal Approach to Software Development.* SpringerVerlag, Berlin, 1990.

[Sha96] N. Shankar. Steps toward mechanizing program transformations using PVS. *Science of Computer Programming*, 26(1–3):33–57, 1996.

[Smi90] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[ST89] Donald Sannella and Andrzej Tarlecki. Toward formal development of ML programs: Foundations and methodology (extended abstract). In *TAPSOFT, Vol.2*, pages 375–389, 1989.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Mass., 1977.

[vHDRS95] Friedrich W. von Henke, Axel Dold, Harald Rues, and Detlef Schwier. Construction and deduction methods for the formal development of software. In *KORSO Book*, pages 239–254, 1995.