MAX-PLANCK-INSTITUT FÜR INFORMATIK

An Abstract Program Generation Logic

David A. Plaisted

MPI–I–94–232 July 1994

INFORMATIK

Im Stadtwald
D 66123 Saarbrücken
Germany

Authors' Addresses

David Plaisted
Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken
Germany
plaisted@mpi-sb.mpg.de

or

Department of Computer Science
CB# 3175, 352 Sitterson Hall
University of North Carolina at Chapel Hill
Chapel Hill, North Carolina 27599-3175
USA
plaisted@cs.unc.edu

Acknowledgements

The comments of David Basin were very helpful in motivating the section on goal-directed program generation and the section on generating programs in specific languages (such as term-rewriting programs). This work was done while the author was on sabbatical and leave of absence at the Max Planck Institute in Saarbrücken, Germany. This work was partially supported by UNC Chapel Hill and by the National Science Foundation under grant number CCR-9108904.

Abstract

We present a system for representing programs as proofs, which combines features of classical and constructive logic. We present the syntax, semantics, and inference rules of the system, and establish soundness and consistency. The system is based on an unspecified underlying logic possessing certain properties. We show how proofs in this system can be systematically converted to programs in a class of abstract logic programming languages including term-rewriting systems and Horn clause logic programs. A number of examples of such logic programming languages and underlying logics are given, as well as some proofs that can be expressed in this system and the corresponding programs.

1 Introduction

We present a programming (meta-)logic PL which is an enrichment of an underlying logic L. This logic, like others, represents programs as proofs, and programs satisfying a certain specification can be extracted from a proof. If the proof is correct, the extracted program is guaranteed to be correct with respect to the specification. Our emphasis is not on automatically constructing programs using a theorem prover, but on representing them in as abstract a manner as possible to facilitate their reuse in different settings. This seems to be of practical importance and also more feasible than automatic program derivation, given the current state of automated reasoning. As hypotheses correspond to subroutines, a proof from a lemma corresponds to an abstract data type that can be instantiated later to obtain a concrete program. However, the system can also be used for (presumably interactive) program generation, and we develop this possibility too. The heart of the system is a systematic method for reasoning about recursion, composition, and fixpoints in an unspecified logic. Although fairly simple, it still may be useful in bringing out and clarifying the connections between different systems, and may be useful for applications. Also, this simplicity has pedagogical advantages. Furthermore, because of the fundamental nature of recursion, composition, and fixpoints, the logic can be considered as another logic of computation. An enriched model of computation (with interleaving or fairness, for example) would require an extension to the inference rules.

The main problem we are trying to deal with is the reuse of programs. Why is it that we have to write the same programs over and over again for different applications, whether a sorting program or a unification program or whatever? The answer may be in part that the algorithms are not represented in an abstract enough manner, and that inessential details obscure the essentials of the algorithm. One goal of the present system is to find an abstract representation for algorithms that will permit them to be used in a variety of settings, without having to be recoded each time. For this purpose, some general methods of combining and instantiating general algorithms for specific applications are mentioned. We note that the techniques presented for this combination of algorithms permit the use of classical logic as opposed to some kind of constructive logic as the main deductive mechanism; this means that the deduction used is often simpler than that in a constructive logic (or, in any case, different).

The logic PL is distinguished from others by its separation between the classical and computational aspects of the logic, and also by its independence from the underlying logic. The extended logic PL introduces new program-construction variables into the underlying logic L and some constructive inference rules for these variables. However, no requirements of constructiveness are imposed on the underlying logic. Furthermore, it only imposes weak demands on the structure of the underlying logic; essentially, first-order logic with a sort structure can be used for many applications. The logic PL does not specify a particular syntax for the programming language. In addition, the logic permits considerable freedom in the underlying computational mechanism, whether deterministic or nondeterministic, functional or relational, terminating or non-terminating and so on. Thus the system is to a large degree independent of the syntax and semantics of the programming language and also from the underlying logic L. In this way we obtain a program generation logic with a high degree of abstractness. This flexibility makes it easy to tailor the logic for specific applications. This also allows for the possibility of translations between different such logics, and we discuss this possibility.

The general idea of the system is to prove statements of the form $(\exists P)A(P)$ where P is a program and A is a property it should satisfy. However, we are interested in constructing modules from which programs can be constructed; thus the statement is rather $(\exists F)A(F)$ where F is a "program constructing function," that is, a function mapping from programs to programs. Given programs that implement certain desired operations, F returns a program that satisfies the top-level specification. Thus we might have a statement of the form $(\exists F)(\forall P_1 \ldots P_n)(A_1(P_1) \wedge \ldots \wedge A_n(P_n) \supset A(F(P_1 \ldots P_n))$ where A_i are the specifications of the subprograms P_i and A is the specification of $F(P_1, \ldots, P_n)$. Given procedures P_i satisfying the specifications A_i, F returns a program $F(P_1, \ldots, P_n)$ satisfying the specification A. Also, F is required to be constructible. However, in the place of the formula $A_1(P_1) \wedge \ldots \wedge A_n(P_n) \supset A(F(P_1 \ldots P_n))$ we allow an arbitrary formula $B(P_1, \ldots, P_n, F(P_1, \ldots, P_n))$ mentioning $P_1, \ldots, P_n$, and F. We can approximately express this as $(\forall P_1 \ldots P_n)(\exists P)B(P_1, \ldots, P_n, P)$, which is similar in form to the specifications used by other program generation systems. The quantifier $(\exists P)$ is required to be constructive, in a sense made precise by the semantics. However, the difference is that $P_1 \ldots P_n$ and P are considered as programs, not inputs and outputs to a program. Though not much different formally from considering $P_1, \ldots, P_n$ and P as inputs and outputs to a program, this difference in point of view has a number of significant consequences. For example, reasoning at the level of programs in this way makes the generation of recursions and fixpoints very natural. We represent this formula $(\forall P_1 \ldots P_n)(\exists P)B(P_1, \ldots, P_n, P)$ using the syntax $[P_1 \ldots P_n \rightarrow P]B(P_1, \ldots, P_n, P)$ where $P_1, \ldots, P_n$ and P are variables representing programs and $[P_1 \ldots P_n \rightarrow P]$ is a new kind of quantifier. The semantics of this requires that P be constructible from $P_1 \ldots P_n$, in a sense made precise by the logic. This syntax allows us to generate programs without explicitly mentioning F, and permits a significant expressive power within a first-order-like system.

The rules of inference of the system consist of several kinds: 1. Those that influence F; these correspond to compositions of programs, recursions, and fixpoint operations, and these are required to be constructive in a certain sense. 2. Those that only influence the specification A, but do not affect F; these need not be constructive. 3. Those that permit an interaction between F and the underlying logic; these for example permit one of the P_i or P to be replaced by an existentially or universally quantified variable in the underlying logic, or permit a quantified variable in the underlying logic to be replaced by a program variable P_i or P. In this way we obtain a combination of constructive and classical operations, and the constructive operations have a simple semantic justification in terms of the operations on F to which they correspond.

The organization of this paper is as follows. First we present the syntax and semantics of the logic, give a set of inference rules for it, and argue for their soundness. Next we present two general proof transformations analogous to the deduction theorem in the propositional calculus. Then we show how proofs in the system may be translated into programs, and also give some mappings between proofs formalized with respect to different underlying logics. We give a number of derived inference rules which facilitate program construction in this system. We discuss the use of derived inference rules for obtaining more efficient programs, and also for facilitating goal-directed program generation. We present a general method for translating proofs in this system into logic programming languages satisfying certain general properties; this includes Horn-clause logic programming and term-rewriting systems as special cases. We illustrate the application of the method to several different underlying logics, too. We show how the system can reason about termination and nontermination, and present some specific algorithms and give their derivations. Finally we comment about higher-order properties of the system and some possible extensions to higher-order reasoning.

We now make some comments about other approaches to program generation. The constructive type theory approaches of [24], Nuprl [5], or the calculus of constructions [12] identify the proof that a term has a type with the proof that a term satisfies a specification. From such a proof, a term (program) in the typed or untyped lambda calculus satisfying the specification can be extracted. Such systems involve constructive higher-order logic and typically synthesize programs from constructive proofs of formula of the form $(\forall x)[P(x) \supset (\exists y)R(x, y)]$, where P is the input assertion and R is the input-output relation. This approach is based on the Curry-Howard isomorphism [20] and the propositions-as-types principle; the latter identifies logical propositions with types whose inhabitants are proofs of the proposition. This approach leads to the synthesis of total functions, although there are some ways to extend this to partial functions. A number of implementations of this idea have been done, including Nuprl [11], Oyster and CLAM [8], and Coq [14].

In [27], a computational interpretation of classical natural deduction is discussed, in which lambda terms may be extracted from proofs; this investigation is continued in [28]. For other papers dealing with computational interpretations of classical logic see [19], [15], [25], and [26]. These systems typically supply a computational interpretation to all classical proofs; in contrast, our system uses classical logic for the part of the proof that does not affect the computation at all.

As for logics dealing with fixed points, we can mention the logic for computable functions of Milner [17]. In [10] a system is given whereby high-level specifications involving least and greatest fixpoints can automatically be transformed into efficient programs. Extensions of Nuprl and the Calculus of Constructions to fixpoints and possibly to nonterminating computations are given, respectively, in [31] and [2].

An early approach to program synthesis is that of [18], who used a resolution theorem prover to derive programs satisfying a specification. It is also possible to synthesize a program by applying a set of transformation rules to a specification. One example of early work in this direction is the deductive synthesis approach of [6]. Another early idea was that of [16], whereby the information contained in a proof can be used to specialize a program to a smaller class of inputs. The TABLOG system of Manna and Waldinger [23] also permits a program to be derived from a specification of the desired input-output behavior. This system is largely first-order (and classical) but with induction rules and rules for specialized theories built in; it uses non-clausal theorem proving in order to facilitate the induction proofs. The Isabelle system [29] permits metalogics to be formalized; it uses a fragment of higher-order intuitionistic logic with higher-order metavariables to formalize the rules of various logics. The Isabelle system can be used to formalize a number of program generation systems [21, 3, 4]. It might also be possible to formalize the present system in Isabelle. In addition to these, there are many other program generation systems which are also based on the idea of proofs as programs or extracting programs from proofs.

2 Syntax

We now return to a discussion of our logic PL and the underlying logic L. The underlying logic is assumed to be some extension of sorted first-order logic. The syntax of the programming logic PL is the syntax of the underlying logic, enriched with individual variables called *program variables*. Some of the sorts of the underlying logic are specified as *program sorts*. Program variables can be of one of the program sorts. These program variables may be individual variables, function or predicate variables, or possibly variables of the lambda calculus, depending on the logic. These variables are intended to represent programs. We use upper case letters (X, Y) for sequences of program variables or individual program variables and lower case (x, y) for variables of the underlying logic. We use lower case letters f, g, h for functions in the underlying logic, letters A, B, C for formulas in the underlying logic, and s, t, u for terms. We note that there is no requirement about which arguments of programs are inputs and which are outputs, or even if they have inputs or outputs; if some convention about inputs and outputs is followed, then the recursion rule and the functional composition rules need to be consistent with this convention.

If X and Y are sequences of program variables, we call $[X \to Y]$ a *program constructor quantifier*. We allow formulas of the form $[X \to Y]A$ where A is a formula of the underlying logic and X, Y are (possibly empty) sequences of program variables and A does not contain any occurrences of the quantifier $[\ldots \to \ldots]$. We can regard A (without occurrences of $\to$) as $[\to]A$. We require X and Y to be disjoint in the quantifier $[X \to Y]$.

By $(\forall X)A$ we mean a sequence of universal quantifications, and similarly for $(\exists X)A$.

3 Semantics

We assume that the underlying logic L is interpreted with respect to a nonempty collection of structures, each having a possibly infinite collection of sorts and a corresponding list $D_1 \ldots D_n \ldots$ of domains and specifying interpretations of the nonlogical symbols as predicates and functions on these domains. Recall that some of the sorts are specified as program sorts. The domains corresponding to the program sorts are called *program domains*. We require that the program domains be non-empty. We extend these structures for the underlying logic to structures for our programming logic PL. This is done by extending the structures to also specify a domain PC of *program construction functions*. We sometimes refer to PC as a sort, and it may have subsorts. We require that the program construction functions have specified arities, that is, the sorts (program domains) of the arguments and result are specified. The program construction functions F map elements $X_1 \ldots X_n$ in specified program domains to an element $F(X_1 \ldots X_n)$ in some specified program domain. Also, we require the following:

1. The projection functions p_i defined by $p_i(X_1 \ldots X_n) = X_i$ are program construction functions.

2. The composition of program construction functions is a program construction function. That is, if $F_i(X_1 \ldots X_m)$ are in PC and G is in PC then $G(F_1(X_1 \ldots X_m), \ldots, F_n(X_1 \ldots X_m))$ is in PC (if the sorts are consistent). (This is to some extent a consequence of property 3, as shown below.) Also, this composition should be constructible.

3. The program construction functions have fixpoints that are also program construction functions. That is, let $F_1(X, Y) \ldots F_n(X, Y)$ be a tuple of program constructing functions, where X and Y are lists of program variables and X has n elements. Suppose that the sorts of $F_i(X, Y)$ and X_i are the same, for $1 \le i \le n$. Then there is tuple $G_1 \ldots G_n$ of program constructing functions such that for all Y, $F_i(G_1(Y), \ldots G_n(Y), Y) = G_i(Y)$, for $1 \le i \le n$. We can write this as $F(G(Y), Y) = G(Y)$, allowing F and G to refer to tuples of program construction functions. We don't require G to be a least fixpoint here, just some fixpoint. Also, we require that some such fixpoint G be constructible from $F_1 \ldots F_n$.

For example, we might have $F_1(X_1, X_2, Y)$ and $F_2(X_1, X_2, Y)$ in PC. Then the fixpoint property would require that there exist functions $G_1(Y)$ and $G_2(Y)$ in PC such that $F_1(G_1(Y), G_2(Y), Y) = G_1(Y)$ and $F_2(G_1(Y), G_2(Y), Y) = G_2(Y)$.

Writing F and G for pairs of functions, we have $F(G(Y),Y) = G(Y)$. We can obtain composition from the fixpoint property as follows: Let H_1 and H_2 be arbitrary unary elements of PC, and let $F(X,Y,Z)$ be $(H_1(X), H_2(Y))$. (Formally, $F_1(X,Y,Z) = H_1(X)$ and $F_2(X,Y,Z) = H_2(Y)$; tuples are not really needed here. However, we do need to be able to compose with projection functions to express F_1 and F_2.) Then F has a fixpoint G, that is, $F(X,G(X)) = G(X)$. Letting $G(X)$ be $(G_1(X), G_2(X))$, we obtain that $F(X, G_1(X), G_2(X)) = (G_1(X), G_2(X)) = (H_1(X), H_2(G_1(X)))$ so $G_1(X) = H_1(X)$ and $G_2(X) = H_2(G_1(X)) = H_2(H_1(X))$. Thus $G(X) = (H_1(X), H_2(H_1(X)))$ and we can compute the composition $H_2 * H_1$ of H_2 and H_1. Similar techniques yield fully general compositions; for this it suffices to take H_1 and F_1 as sequences of m functions and X as a sequence of n program variables and H_2 of arity m. Thus G_1 is a sequence of m functions, F and G are sequences of $m+1$ functions, $H_2 * H_1(X)$ is $H_2(H_{11}(X), \ldots, H_{1m}(X))$, and we can compute a general composition.

We use F, G, H for program constructing functions and also later on for program variables. We call a structure satisfying the above three properties, a *programming structure*. In order to show soundness of the logic, it is necessary that the formulas of PL be interpreted relative to a set of structures in which the above properties hold. We note that for recursive functions, fixpoints as in 3 exist by the recursion theorem. The requirement of constructibility is difficult to test or even understand in a general context as above. We will make this more concrete in our discussion of the generation of programs in specific languages.

We interpret a formula $[X \to Y]A[X,Y]$ where $|X| = m$ and $|Y| = n$ as follows: There exists in PC a tuple of n program construction functions $F_1 \ldots F_n$ with m arguments such that for all X in the respective program domains, $A[X, F_1(X), \ldots, F_n(X)]$. We can express this as the following formula in PL, where we add variables F_i of sort PC (or its subsorts): $(\exists F_1 \ldots F_n \in PC)(\forall X)A[X, F_1(X), \ldots F_n(X)]$. However, we note that this formula does not fully express the semantics. Not only do we require that the F_i exist, but the proof of this must be constructive. Thus we have a mixture of classical and constructive logic, with the constructive part restricted to the program constructing functions. Note also that we do not prove constructively the existence of outputs to a program. Nor do we prove constructively the existence of a program. Rather, we prove constructively the existence of the program constructing functions F_i. The reason is that we want to develop a set of building blocks that can frequently be reused and combined to form desired programs. The program constructing functions are such building blocks, because they map programs to programs, and by specifying their (program) arguments they can be instantiated to obtain particular programs having specified properties.

Our system can be applied to domains in which least fixpoints may not exist, or may be difficult to express, or for which the proper concept of a least fixpoint may be obscure. However, domains in which fixpoints and least fixpoints exist typically may be expressed as *complete partial orders*. That is, there is a partial ordering $<_d$ defined on each sort of program construction function and an element $\perp$ such that $\perp \leq_d x$ for all x. Also, if $x_1, x_2, x_3, \ldots$ is a sequence with $x_1 \leq_d x_2 \leq_d x_3 \ldots$ then the least upper bound $\bigsqcup_i x_i$ exists. A function f is *monotone* if $x \leq_d y$ implies $f(x) \leq_d f(y)$ and *continuous* if for all monotone sequences $x_1, x_2, x_3, \ldots$ we have that $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$. It turns out that computable functions typically correspond to continuous functions on a domain, and so we can take PC to be the set of continuous functions in the appropriate domains. All continuous functions are monotone, and the composition of continuous functions is continuous. We can extend the ordering to functions by $f \leq_d g$ if for all x, $f(x) \leq_d g(x)$. A value x is a *fixed point* of f if $f(x) = x$. It is known that continuous functions f have the least fixed point $\bigsqcup_i f^i(\perp)$. For functions $f(x_1, \ldots, x_n)$ with more than one argument, we say they are continous if $f(\bigsqcup_i x_{1i}, \ldots, \bigsqcup_i x_{ni}) = \bigsqcup_i f(x_{1i}, \ldots, x_{ni})$. In this case, we can take fixpoints with respect to one of the arguments. For example, to find a least y such that $f(x, y) = y$ (or more precisely, a least function $g(x)$ such that for all x, $f(x, g(x)) = g(x)$), we can take $g(x) = \bigsqcup_i (\lambda y.f(x, y))^i(\perp)$. Such a function g will also be continuous. This can be generalized to more than one function and more than one argument. A domain is called *flat* if $x <_d y$ implies $x = \perp$. A function f is said to be *strict* if $f(\ \ldots,\ \perp,\ \ldots\ ) = \perp$. The intuition is that if $x <_d y$ then x is "less defined" than y; $\perp$ is the totally undefined element, in the sense that nothing is known about its value. It is customary therefore to let $\perp$ represent a nonterminating computation that returns no information.

It may be that the original semantics is not given in terms of programming structures. For example, programs may be interpreted as character strings and an evaluation function may be used to extract their behavior. In this case, the fixpoint property will not directly hold, since typically the fixpoint property applies to the semantics of a program and not to its syntax. However, we can change the semantics so that the meaning (interpretation) of a program is its behavior. That is, if a program P appears only in the context $f_{eval}(P, x_1,\ \ldots,\ x_n)$ where f_{eval} is some kind of evaluation function, then P can be interpreted as the function from $x_1\ \ldots\ x_n$ to $f_{eval}(P, x_1\ \ldots\ x_n)$. In this modified semantics, the fixpoint property may hold.

4 Inference Rules

As usual, we can rename bound variables subject to capture. Also, duplicates in X and Y can be eliminated, and variables in X and Y can be permuted. The functional composition rules and the least fixpoint rules may be included or omitted, depending on the underlying logic.

$$\frac{[X, U \to Y, Z]A[X, U, Y, Z]}{[X \to Y, Z]A[X, Y, Y, Z]} \quad (\textit{Recursion rule})$$

where U and Y have the same sort.

For functional programming:

(Functional composition rule 1)

$$[F_1\ \ldots\ F_m \to G](\forall z_1\ \ldots\ z_n)G(z_1\ \ldots,\ z_n) = T(F_1,\ \ldots,\ F_m, z_1,\ \ldots,\ z_n)$$

assuming that the program variables F_i and G are of functional type, where T is a well-formed term involving only the variables z_i and applications of function variables F_i to appropriate numbers and sorts of arguments.

(Functional composition rule 2)

$$[P, F_1\ \ldots\ F_m \to Q](\forall z_1\ \ldots\ z_n)(Q(z_1\ \ldots\ z_n) \equiv P(F_1(z_1\ \ldots\ z_n),\ \ldots,\ F_n(z_1\ \ldots\ z_n)))$$

Here P is a program variable that is a predicate in the underlying logic and the F_i are function variables.

(Least fixpoint rule, where assumed)

$$\frac{[X, U \to Y, Z]A[X, U, Y, Z] \qquad (\forall XU)(\exists! Y)(\exists Z)A[X, U, Y, Z]}{[X \to Y, Z](A[X, Y, Y, Z] \wedge (\forall Y')((\exists Z')A[X, Y', Y', Z']) \supset Y \leq_d Y')}$$

Here $\exists!$ means "There exists unique" and $<_d$ is a domain ordering.

(Underlying logic rule)

$$\frac{[X \to Y]A}{[X \to Y]B} \quad if \ \vdash (\forall X, Y)(A \supset B) \ in\ the\ underlying\ logic\ L$$

We note that this derivation in the underlying logic need not be constructive.

$$\frac{[X, Y \to]A}{[Y \to](\forall X)A} \quad (Left\ elimination\ rule)$$

$$\frac{[X \to Y, Z]A}{[X \to Y](\exists Z)A} \quad (Right\ elimination\ rule)$$

This rule can be used to eliminate excess output variables.

$$\frac{[X \to Y](\forall Z)A}{[Z, X \to Y]A} \quad (Left\ introduction\ rule;\ Z, Y\ must\ be\ disjoint)$$

This rule can be used to add excess input variables.

$$\frac{[X \to Y](\forall Z)A}{[X \to Y, Z]A} \quad (Right\ introduction\ rule;\ Z, X\ must\ be\ disjoint)$$

This rule is valid since at least one Z is computable. This rule can be used to add excess output variables.

$$\frac{[X \to Y]A \quad [X \to Z]B}{[X \to Y, Z](A \wedge B)} \quad (Conjunction\ rule;\ if\ no\ captures\ and\ Y, Z\ are\ disjoint)$$

We illustrate recursion as follows: Suppose X and Y have the same type. Then the formula $[X \to Y, Z]A[X, Y, Z]$ represents a method of obtaining programs Y and Z from X. We then have the following derivation:

1.	$[X \to Y, Z]$	$A[X, Y, Z]$	(given)
2.	$[\to Y, Z]$	$A[Y, Y, Z]$	(1, recursion rule)

The conclusion represents a program that has the input X and the output Y the same. We can think of this as replacing the input programs X by a recursive call to the program-constructing operation implicit in 1. We will illustrate this below with a derivation of the (recursive) factorial function using an application of the recursion rule.

We also have a stronger version of the fixpoint rule which only guarantees the existence of a least fixpoint for input programs X and U such that Y is unique, in the formula $A[X, U, Y, Z]$:

$$\frac{[X, U \to Y, Z]A[X, U, Y, Z]}{[X \to Y, Z](A[X, Y, Y, Z] \wedge (\forall Y')((\forall W)(\exists Z')A[X, Y', W, Z'] \equiv (W = Y')) \supset (Y \leq_d Y'))}$$

4.1 Derived Rules

We give some example proofs in this system.

1.	$[X, W \to Y]$	$A[X, W, Y]$	
2.	$[X, W \to Y]$	$(\forall\ Z)A[X, W, Y]$	where Z does not appear in A (by the underlying logic rule)
3.	$[X, W \to Y, Z]$	$A[X, W, Y]$	(right introduction) (Z disjoint with W and X)
4.	$[W \to Y, Z]$	$A[Z, W, Y]$	(recursion rule)
5.	$[W \to Y, X]$	$A[X, W, Y]$	(renaming rule) (X, W disjoint)

The intuition is that in 5, X are specific programs and Y are programs obtained from them using the mapping of 1. Thus we get the derived inference rule

$$\frac{[X, W \to Y]A}{[W \to X, Y]A} \quad (W, X \ disjoint)$$

Here is a proof of another derived rule:

1.	$[X, W \to Y]$	$A(X, Y)$	
2.	$[X, W \to Y]$	$(\forall Z)A(X, Y)$	(underlying logic rule) where the sorts of X an Z are the same
3.	$[X, W \to Y, Z]$	$A(X, Y)$	(right introduction rule) (Z disjoint from X and W)
4.	$[W \to Y, Z]$	$A(Z, Y)$	(recursion rule on X and Z)
5.	$[W \to Y]$	$(\exists Z)A(Z, Y)$	(right elimination)
6.	$[W \to Y]$	$(\exists X)A(X, Y)$	(renaming)

Thus we obtain the rule

$$\frac{[X, W \to Y]A(X, Y)}{[W \to Y](\exists X)A(X, Y)}$$

which allows input variables to be replaced by existentially quantified variables. We now show how unnecessary input variables can be eliminated:

1.	$[X, Y \to Z]$	$A[X, Z]$	where Y do not appear in A
2.	$[X \to Y]$	$(\exists W)A[X, Z]$	(using the above derived rule)
3.	$[X \to Y]$	$A[X, Z]$	(underlying logic rule)

Thus we have the derived inference rule

$$\frac{[X, Y \to Z]A[X, Z]}{[X \to Y]A[X, Z]}$$

where Y do not appear in A. We show how separate program forming operations can be combined "in parallel". Assume (W,Y), (Z,X), and (Y,Z) are disjoint.

1.	$[X \to Y]$	$A[X, Y]$	given
2.	$[W \to Z]$	$B[W, Z]$	given
3.	$[X \to Y]$	$(\forall W)A[X, Y]$	(1, underlying logic rule)
4.	$[W \to Z]$	$(\forall X)B[W, Z]$	(2, underlying logic rule)
5.	$[X, W \to Y]$	$A[X, Y]$	(3, left introduction rule) (Y,W disjoint)
6,	$[X, W \to Z]$	$B[W, Z]$	(4, left introduction rule) (Z,X disjoint)
7.	$[X, W \to Y, Z]$	$(A[X, Y] \wedge B[W, Z])$	(5,6, conjunction rule) (Y,Z disjoint)

Thus we obtain the following inference rule:

$$\frac{[X \to Y]A \quad [W \to Z]B}{[X, W \to Y, Z](A \wedge B)} \quad (if\ no\ captures\ and\ Y, Z\ are\ disjoint\ and\ W, Y\ and\ Z, X\ are\ disjoint)$$

We call this the combination rule. We now illustrate a kind of program composition.

1.	$[X \to Y]$	$A[X,Y]$	(given)
2.	$[Y \to Z]$	$B[Y,Z]$	(given) (note that Y,Z are disjoint)
3.	$[V \to Z]$	$B[V,Z]$	(renaming, 2) (so that V,Z disjoint)
4.	$[X,V \to Y,Z]$	$A[X,Y] \wedge B[V,Z]$	(combination rule) (Y,Z and X,Z and V,Y disjoint)
5.	$[X \to Y,Z]$	$A[X,Y] \wedge B[Y,Z]$	(recursion rule)
6.	$[X \to Z]$	$(\exists Y)A[X,Y] \wedge B[Y,Z]$	(right elimination rule)

In this way we derive the following rule:

$$\frac{[X \to Y]A \quad [Y \to Z]B}{[X \to Z](\exists Y)A \wedge B} \ (assuming\ X, Z\ disjoint)$$

We call this the program composition rule. We would like to have the following more general rule too:

$$\frac{[X \to Y]A \quad [Y,W \to Z]B}{[X,W \to Z](\exists Y)A \wedge B} \ (assuming\ X, Z\ and\ W, Y\ disjoint)$$

For this purpose we modify the above proof as follows:

1.	$[X \to Y]$	$A[X,Y]$	(given)
2.	$[Y,W \to Z]$	$B[Y,W,Z]$	(given) (note that Y, Z and W, Z are disjoint)
3.	$[V,W \to Z]$	$B[V,W,Z]$	(renaming, 2) (so that V, Z are disjoint)
4.	$[X,V,W \to Y,Z]$	$A[X,Y] \wedge B[V,W,Z]$	(combination rule) (Y, Z and X, Z and V, Y and W, Y disjoint)
5.	$[X,W \to Y,Z]$	$A[X,Y] \wedge B[Y,W,Z]$	(recursion rule)
6.	$[X,W \to Z]$	$(\exists Y)A[X,Y] \wedge B[Y,W,Z]$	(right elimination rule)

We now derive a kind of converse of the conjunction rule:

1.	$[X \to Y,Z]$	$(A \wedge B)$	(assumed)
2.	$[X \to Y]$	$(\exists Z)(A \wedge B)$	(right elimination)
3.	$[X \to Y]$	$A \wedge (\exists Z)B$	(underlying logic rule, assuming Z does not occur in A)
4.	$[X \to Y]$	A	(underlying logic rule)

Thus we obtain the following derived rule:

$$\frac{[X \to Y,Z](A \wedge B)}{[X \to Y]A} \ (assuming\ Z\ does\ not\ occur\ in\ A)$$

This permits us to separate out individual program constructing functions. The following rule (identifying inputs) is sometimes convenient. Assuming the identity is computable (which follows from the first functional composition rule), we have

1.	$[X \to Y]$	$(Y = X)$	
2.	$[X \to Z]$	$(Z = X)$	
3.	$[X \to Y,Z]$	$(Y = X \wedge Z = X)$	(1,2, conjunction rule)
4.	$[Y,Z \to W]$	$A(Y,Z,W)$	(assumption)
5.	$[X \to W]$	$(\exists Y,Z)(Y = X \wedge Z = X \wedge A(Y,Z,W))$	(3,4, composition rule)
6.	$[X \to W]$	$A(X,X,W)$	(5, underlying logic rule)

Thus we derive the rule

$$\frac{[Y,Z \to W]A(Y,Z,W)}{[X \to W]A(X,X,W)}$$

which permits us to make two input variables equal. We now derive another version of the functional composition rule:

$$\frac{[\rightarrow H](H_1 = k_1 \wedge \ \ldots \ \wedge H_n = k_n)}{[F \rightarrow G](\forall z_1 \ \ldots \ z_n)G(z_1, \ \ldots, \ z_n) = T(F, k, z_1, \ \ldots \ , z_n)}$$

assuming that the program variables F and G are of functional type, where T is a well-formed term involving only the variables z_i and applications of function variables F_i and k_j to appropriate numbers and sorts of arguments, where F is $(F_1 \ \ldots \ F_m)$ and H is $(H_1 \ \ldots \ H_n)$ and k is $(k_1 \ \ldots \ k_n)$. We call this the functional composition rule with individual functions. The intention is that the k_i are known individual function symbols in the underlying logic and they are also known to be computable. Then we are allowed to use these function symbols in the functional composition rule. The proof is as follows, letting Z abbreviate $z_1 \ \ldots \ z_n$ and $H = k$ abbreviate $H_1 = k_1 \wedge \ \ldots \ \wedge H_n = k_n$:

1.	$[F, H \rightarrow G]$	$(\forall z)G(z) = T(F, H, z)$	(first functional composition rule)
2.	$[\rightarrow H]$	$(H = k)$	(assumption)
3.	$[F \rightarrow G]$	$(\exists H)(H = k \wedge (\forall z)G(z) = T(F, H, z))$	(program composition rule,1,2)
4.	$[F \rightarrow G]$	$(\forall z)G(z) = T(F, k, z)$	(3, underlying logic rule)

We now show how the functional composition rule can be extended to allow recursion, with abbreviations as above and G a sequence of program variables and T a sequence of terms:

$$\frac{[\rightarrow H](H = k)}{[F \rightarrow G](\forall z)G(z) = T(F, k, G, z)}$$

where T is a sequence of well-formed terms involving the function symbols appearing in the sequences F,k, and G. Since G is defined in terms of itself, we have a recursive definition. We call this the recursive functional composition rule. The proof is as follows:

1.	$[\rightarrow H]$	$(H = k)$	(assumed)
2.	$[F \rightarrow H]$	$(\forall F')(H = k)$	(underlying logic rule)
3.	$[F, F' \rightarrow H]$	$(H = k)$	(left introduction rule)
4.	$[F, F' \rightarrow G]$	$(\forall z)G(z) = T(F, F', k, z)$	(3, a sequence of applications of the functional composition rule with individual functions)
5.	$[F \rightarrow G]$	$(\forall z)G(z) = T(F, G, k, z)$	(4, recursion rule)

This rule permits a number of functions to be defined in terms of one another using mutual recursion and known computable individual functions. This rule is fairly powerful and permits many functions to be derived in one step. We later comment on how this can be controlled.

Note that from these rules we can derive the following:

$$[X \rightarrow Y]true$$

$$\frac{(\forall X, Y)A}{[X \rightarrow Y]A}$$

$$\frac{(\forall X)A}{[\rightarrow X]A}$$

$$\frac{[X \rightarrow Y]A}{(\forall X)(\exists Y)A}$$

5 Soundness of the Logic

Using this semantics we can show that the rules of inference are sound (and constructive in the functions F_i). For this we have to consider the following rules:

renaming rule	left elimination rule
permuting rule	right elimination rule
eliminating duplicates	introduction rule
recursion rule	right introduction rule
functional composition rules	conjunction rule
underlying logic rule	least fixpoint rule

We call $(\exists F_1 \ldots F_n \in PC)(\forall X)A[X, F_1(X), \ldots, F_n(X)]$ the *semantic formula* for $[X \to Y]A$. We abbreviate this formula as $[(F_1 \ldots F_n) : X \to Y]A[X, Y]$. We often use F as an abbreviation for a list $(F_1 \ldots F_n)$ of program constructing functions; then we have $[F : X \to Y]A[X, Y]$ as an abbreviation for this formula. We can also write it as $(\exists F)(\forall X)A[X, F[X]]$. Yet another representation is $(\forall X, Y)(F[X] = Y \supset A[X, Y])$. We now show that the rules are sound. As for the renaming rule, since we can rename X in the formula $(\forall X)A[X, F_1(X), \ldots, F_n(X)]$, we can also rename X in $[X \to Y]A$. We can rename Y since the variables Y do not explicitly appear in the semantic formula. Duplicates in X can be eliminated since $(\forall X)(\forall X)B$ is equivalent to $(\forall X)B$. The X variables can be permuted since universal quantifiers can be permuted, and because the projection functions (and compositions) are computable. The Y variables can be permuted since the existential quantifiers for $F_1 \ldots F_n$ can be permuted. Duplicates in the Y variables can be eliminated since duplicate existential quantifiers can be eliminated. None of these rules introduce new F_i, so constructibility of the F_i is preserved. Some of these rules permute the F_i or their arguments but these transformations are all constructible.

The recursion rule is justified by the fact that the program construction functions have constructible fixpoints, property 3 above. In particular, assume $[X, U \to Y, Z]A[X, U, Y, Z]$. The corresponding semantic formula is $(\exists F, G \in PC)(\forall X, U)$ $A[X, U, F(X, U), G(X, U)]$. Let F and G refer to particular elements of PC, so we have now $(\forall X, U)$ $A[X, U, F(X, U), G(X, U)]$. Let $H(X)$ be a fixpoint of F, that is, $F(X, H(X)) = H(X)$. Instantiating the semantic formula, we obtain $(\forall X)$ $A[X, H(X), F(X, H(X)), G(X, H(X))]$. Since $F(X, H(X)) = H(X)$, we have $(\forall X)$ $A[X, H(X), H(X), G(X, H(X))]$. Since functions in PC have constructible fixpoints, $H(X)$ is also in PC. Also, since compositions of elements of PC are in PC, we have that the function $G'(X) = G(X, H(X))$ is also in PC. Therefore we obtain the formula $(\forall X)A[X, H(X), H(X), G'(X)]$, and from it $(\exists H, G' \in PC)(\forall X)A[X, H(X), H(X), G'(X)]$, which is the semantic formula for $[X \to Y, Z]$ $A[X, Y, Y, Z]$.

We now consider the functional composition rule. In a functional setting, we typically choose the program domains to include all (well-sorted) compositions of elements of the program domains, justifying the functional composition rules. The second rule interfaces functions and predicates. Also, such compositions are constructible. However, these rules need not be sound for all programming logics PL. The underlying logic rule is sound since if $(\forall X, Y)$ $(A[X, Y] \supset B[X, Y])$ in the underlying logic then $(\forall X)$ $A[X, F_1(X), \ldots, F_n(X)]$ implies $(\forall X)$ $B[X, F_1(X), \ldots, F_n(X)]$, hence $(\exists F_1 \ldots F_n \in PC)(\forall X)$ $A[X, F_1(X), \ldots, F_n(X)]$ implies $(\exists F_1 \ldots F_n \in PC)(\forall X)$ $B[X, F_1(X), \ldots, F_n(X)]$, so $[X \to Y]A$ implies $[X \to Y]B$. This does not change the F_i, so we preserve constructibility of the F_i.

The left elimination rule is sound since the semantic formula for $[X, Y \to]A$ is $(\forall X, Y)A$. The right elimination rule is sound since some of the expressions $F_i(X)$ can be replaced by existentially quantified variables in the semantic formula.

The left introduction rule is sound by properties of universal quantifiers in the underlying logic. We are giving the semantic functions extra arguments, which is allowed because projections are computable and composition is computable. The right introduction rule is sound since there is at least one computable program constructing function of each sort and we can choose this to be the F_i corresponding to Z. These rules all transform the F_i in constructible ways, often not at all.

We consider the conjunction rule:

$$\frac{[X \to Y]A \quad [X \to Z]B}{[X \to Y, Z](A \wedge B)} \quad (\mathit{Conjunction\ rule;\ if\ no\ captures\ and}\ Y, Z\ \mathit{are\ disjoint})$$

This rule is valid because existential quantification (for the F_i) can be pushed inside conjunction when the existentially quantified variables only appear in one conjunct. That is, $(\exists F)(\forall X)A[X, F[X]] \wedge (\exists G)(\forall X)B[X, G[X]]$ is equivalent to $(\exists F, G)((\forall X)A[X, F[X]] \wedge (\forall X)B[X, G[X]])$, or, to $(\exists F, G)\ (\forall X)\ (A[X, F[X]] \wedge B[X, G[X]])$, which is $[(F, G) : X \to Y, Z](A[Y] \wedge B[Z])$. This rule combines two sets of program construction functions, an operation which is constructible.

We now consider the least fixpoint rule:

$$\frac{[X, U \to Y, Z]A[X, U, Y, Z] \quad (\forall XU)(\exists! Y)(\exists Z)A[X, U, Y, Z]}{[X \to Y, Z](A[X, Y, Y, Z] \wedge (\forall Y')((\exists Z')A[X, Y', Y', Z']) \supset Y \leq_d Y')}$$

Here $(\exists! Y)$ means "there exists a unique Y." For this rule, $\leq_d$ must be a domain ordering satisfying appropriate conditions, that is, least fixpoints of continuous functions exist and are constructible. These are assumptions that are often satisfied in practice. Also, we need to assume that all constructible program constructing functions are continuous. From the hypothesis $[X, U \to Y, Z]A[X, U, Y, Z]$ we have that there are tuples F and G of program contructing functions such that $(\forall XU)A[X, U, F[X, U], G[X, U]]$. We then know that for each X there is a least Y such that $F[X, Y] = Y$. We assume that least fixpoints are constructable, so let H be an element of PC such that $F[X, H[X]] = H[X]$ and such that $H[X]$ is a minimal such element. Thus we have $(\forall X)\ A[X, H[X], F[X, H[X]], G[X, H[X]]]$, that is, $(\forall X)\ A[X, H[X], H[X], G[X, H[X]]]$. Thus we have $[X \to Y, Z]A[X, Y, Y, Z]$, reasoning as in the recursion rule. Now, if $(\exists Z')A[X, Y', Y', Z']$, then from $A[X, Y', F[X, Y'], G[X, Y']]$ we know that Y' is $F[X, Y']$, since we are given the hypothesis $(\forall XU)(\exists! Y)\ (\exists Z)A[X, U, Y, Z]$. Since Y' is $F[X, Y']$, Y' is a fixpoint of F. Letting Y be the least fixpoint $H[X]$ of F, we have that $A[X, Y, Y, Z]$ and $Y \leq_d Y'$. This is what is needed. The proof for the stronger version of the fixpoint rule is similar, except that we only can conclude $Y \leq_d Y'$ when the uniqueness assumption holds. We note that domains permit us to express nontermination using "bottom" ($\perp$, or undefined) if desired. Thus we can reason about termination and nontermination within this formalism. Also, it is not necessary that all programs terminate; we can represent nonterminating programs.

5.1 Consistency

We note that if we interpret the formula $[X \to Y]A$ as $(\forall X)(\exists Y)A$, then all inference rules are sound in the underlying logic except possibly the recursion rule and the functional composition rules. Therefore, if the underlying logic is sound, any proof not using the recursion rule or the functional composition rules is sound. For the full logic, we note that consistency is not a trivial matter. Consider the formula $[X \to Y](Y = X + 1)$. If we apply the recursion rule we obtain $[\to Y](Y = Y + 1)$. Applying right elimination we get $(\exists Y)(Y = Y + 1)$. Using the underlying logic rule we obtain $0 = 1$. However, we want to have addition by 1 computable since the logic is supposed to capture computability. Therefore we must take care to ensure consistency in some other way. For this, we need to have domains in which least fixpoints exist. This means that the integers would be extended with a "bottom" element $\perp$ and we then have $\perp = \perp + 1$ removing the inconsistency. So we can only say $Y - Y = 0$ if $Y \neq \perp$, which is annoying but manageable, and this prevents the above derivation of $0 = 1$. Note that such considerations only apply to values produced by computations; in the underlying logic, if we know that Y is an integer then we can use the identity $Y - Y = 0$. This introduction of fixpoints causes other problems. For example, in the underlying logic, we would like to have the rule $(\forall x)(P(x) \vee \neg P(x))$. This implies $P(\perp) \vee \neg P(\perp)$. However, computable functions are typically taken to be monotone. This means that if P is computable, then $P(\perp) \supset (\forall x)P(x)$. This would imply that if P and $\neg P$ are computable and nontrivial, we cannot have $(\forall x)(P(x) \vee \neg P(x))$. It would be highly unpleasant to give up the rule $(\forall x)(P(x) \vee \neg P(x))$. We could develop special rules for "$\perp$"; for example, we could replace $(\forall x)A$ by $(\forall x)((x \neq \perp) \supset A)$ everywhere. But the domain structures can be more complicated than this, and so such an approach is not in general sufficient. Our approach is to say that computability is explicitly treated by the logic PL, and that the underlying logic need not be concerned with it. Therefore, if P is computable, $\neg P$ may be uncomputable, but we can still use it and have the axiom $(\forall x)(P(x) \vee \neg P(x))$. Therefore we may have $P(x)$ for some x and not $P(\perp)$; this implies that $\neg P$ is not monotone, and therefore not computable. Later we will give a general method for "coercing" values of computed predicates to Booleans, so that the underlying logic need not deal with "bottom" as a value of a predicate.

If we interpret the formula $[X \rightarrow Y]A$ as the semantic formula $(\exists F_1 \ \dots \ F_n \in PC)(\forall X)A[X, F_1(X), \ \dots, \ F_n(X)]$ then again all rules of inference are sound in the underlying logic except the functional composition rules and the recursion rule. However, if we assume that the underlying logic has at least one programming structure (that is, satisfying the above three properties about program constructing functions), then all rules of inference are sound including the recursion rule. If the functional composition rules are used, then corresponding properties of the program domains and the sort PC must be assumed for the underlying logic. In this way we can translate any proof involving formulae of PC into a proof in the underlying logic. In order to derive both a formula W and its negation, W must not have a program constructor quantifier, since none of the formulas have negations outside of the program constructor quantifier $[X \rightarrow Y]$. Therefore W must be a formula in the underlying logic. However, these PL derivations of W and its negation can be translated into derivations in the underlying logic using the semantic formulas in place of the formulas of PL. This would imply that the underlying logic were inconsistent. However, we assumed that the underlying logic had at least one structure satisfying the specified properties, that is, at least one model M. Also, all rules of inference, translated in this way, are sound. Thus M would have to satisfy both W and its negation, which is not possible. We note again, however, that this translation of formulas into the corresponding semantic formulas does not fully capture the semantics of the logic, since the program constructing functions must be derived constructively.

6 General Deduction Theorems

Some kind of general deduction theorems can give us more flexibility in the way rules are written. These also may make some proofs easier to write (and derive). We derive such rules here syntactically and semantically. For this we use $\vdash_{PL}$ to indicate derivability in the logic PL.

6.1 First deduction principle

D_1. If we can show

$$\frac{[\rightarrow Y_1]A_1 \quad \Gamma}{[X \rightarrow Y]A}$$

(that is, $\Gamma,\ [\rightarrow Y_1]A_1 \vdash_{PL} [X \rightarrow Y]A$) then it follows that

$$\frac{\Gamma}{[Y_1, X \rightarrow Y](A_1 \supset A)}$$

(that is, $\Gamma \vdash_{PL} [Y_1, X \rightarrow Y](A_1 \supset A)$) assuming that Y_1 and Y are disjoint. We can give a (semantic) validity argument as follows. We express Γ in more detail as follows:

$$\frac{[\rightarrow Y_1]A_1(Y_1) \quad [X_2 \rightarrow Y_2]A_2(X_2, Y_2) \ \ldots\ [X_n \rightarrow Y_n]A_n(X_n, Y_n)}{[X \rightarrow Y]A(X, Y)}$$

We show that

$$\frac{[X_2 \rightarrow Y_2]A_2(X_2, Y_2) \ \ldots\ [X_n \rightarrow Y_n]A_n(X_n, Y_n)}{[Y_1, X \rightarrow Y]A_1(Y_1) \supset A(X, Y)}$$

We do this as follows. From the hypotheses we have F_i such that $A_1(F_1)$ and $(\forall X_i)A_i(X_i, F_i(X_i))$, $i > 1$. Then $G(F_1 \ \ldots\ F_n)$ is such that $(\forall X)A(X, G(X))$. Thus $(\forall X)A(X, G(F_1 \ \ldots\ F_n)(X))$. We can write this as $(\forall X)A(X, G(F_2 \ \ldots\ F_n)$ $(F_1, X))$. Then we have that $\wedge_i(\forall X_i)A_i(X_i, F(X_i)) \supset ((\forall X, Y_1)A_1(Y_1) \supset A(X,$ $G(F_2 \ \ldots\ F_n)\ (Y_1, X)))$. This corresponds to the inference rule

$$\frac{[X_2 \rightarrow Y_2]A_2(X_2, Y_2) \ \ldots\ [X_n \rightarrow Y_n]A_n(X_n, Y_n)}{[Y_1, X \rightarrow Y]A_1(Y_1) \supset A(X, Y)}$$

And of course if we have Y_1 in the conclusion, it can be moved back as a hypothesis using the composition rule, so we can go from the lower form of the inference rule to the upper form.

6.2 Second deduction principle

D_2. If

$$\frac{[W \rightarrow Y_1]A_1 \quad \Gamma}{[Z \rightarrow Y]A}$$

then

$$\frac{[X, W \rightarrow Y_1]C(X) \supset A_1 \quad \Gamma}{[X, Z \rightarrow Y]C(X) \supset A}$$

Note that A_1 only mentions Y_1, not X. This can be shown valid by a simple syntactic argument. Suppose we have the upper inference rule. Suppose also that we have $[X, W \rightarrow Y_1]C(X) \supset A_1$ and Γ. Now, we want to show that $[X, Z \rightarrow Y]C(X) \supset A$ can be derived. By the first deduction principle, it suffices to show that from $[\rightarrow X]C(X)$ we can derive $[Z \rightarrow Y]A$. From $[\rightarrow X]C(X)$ and $[X, W \rightarrow Y_1]C(X) \supset A_1$ we can derive $[W \rightarrow Y_1]A_1$ by the composition rule. Then by the upper inference rule we can derive $[Z \rightarrow Y]A$. This is all that is required. We now show that the lower rule (for all C) implies the upper one. For this, it suffices to let $C(X)$ be "true." Then the X variables in the hypothesis and conclusion become superfluous, and can be eliminated, leading to the upper rule.

6.3 Syntactic proofs

We now give syntactic proofs of the first deduction principle. We rewrite this principle in the following way (D_1'):

If

$$\frac{[\to X]A(X) \quad \Gamma}{[Y \to Z]B(Y,Z)}$$

then

$$\Gamma \vdash_{PL} [X, Y \to Z]A(X) \supset B(Y,Z)$$

assuming that X and Z are disjoint. This needs to be shown for each of the inference rules of PL. Then we show D_2': if there is an inference rule of the form

$$\frac{[X \to Y]A(X,Y) \quad \Gamma}{[U \to V]B(U,V)}$$

then we have

$$\Gamma,\ [W, X \to Y]C(W) \supset A(X,Y) \vdash_{PL} [W, U \to V]C(W) \supset B(U,V)$$

where W and Y and W and V are disjoint. Together these suffice to prove the first (hence both) deduction principles, by induction on proof size. We note that renamings of variables in the conclusion are not significant and are permitted when necessary. We start with D_1'. We consider the renaming variables rule. The only case of concern to us is the following:

$$\frac{[\to X]A(X)}{[\to Y]A(Y)}$$

We need to show that $[X \to Y]A(X) \supset A(Y)$ (when X and Y are disjoint). This follows because the identity is computable (from the first functional composition rule).

Now we consider the rule that permits us to permute variables:

$$\frac{[\to X]A(X)}{[\to Y]A(X)}$$

where Y is a permutation of X. We need to show that $[X \to Y]A(X) \supset A(X)$. This is not allowed, since X and Y have common variables and this is not permitted. So let Y' and X' be consistent renamings of Y and X (since the names of the variables is not significant). Then we want to show that $[X \to Y']A(X) \supset A(X')$. By the permutation of variables rule, it suffices to show $[X \to X']A(X) \supset A(X')$. This follows from $[X \to X'](X' = X)$; by the underlying logic rule we obtain $[X \to X'](X' = X) \wedge A(X) \supset A(X)$. By the underlying logic rule again we obtain $[X \to X']A(X) \supset A(X')$.

The recursion rule and least fixpoint rules are not relevant because the hypothesis is not of the right form. The functional composition rules are not applicable because they have no hypothesis. Consider the underlying logic rule; in the form of concern to us it is

$$\frac{[\to X]A(X)}{[\to X]B(X)}$$

where $A(X) \vdash B(X)$ in the underlying logic. To separate the variables, we make this

$$\frac{[\to X]A(X)}{[\to Y]B(Y)}$$

and desire to show that $[X \to Y]A(X) \supset B(Y)$. This follows again from the rule $[X \to Y](Y = X)$ by the underlying logic rule (assuming equality replacement is allowable there). The remaining rules are handled by similar arguments.

We now go back and show D_2' for each inference rule. Suppose we have

$$\frac{[X \to Y]A(X,Y) \quad \Gamma}{[U \to V]B(U,V)}$$

and the conclusion is obtained by renaming variables; then Γ is empty and the rule is of the form

$$\frac{[X \to Y]A(X,Y)}{[U \to V]A(U,V)}$$

We want to show that $[Z, X \to Y]C(Z) \supset A(X,Y) \vdash_{PL} [Z, U \to V]C(Z) \supset A(U,V)$. But this is obtained directly by renaming variables. The permuting variables rule is equally direct.

Consider the recursion rule:

$$\frac{[X, U \to Y, Z]A(X,U,Y,Z)}{[X \to Y, Z]A(X,Y,Y,Z)}$$

From this general schema we also immediately have

$$\frac{[W, X, U \to Y, Z]C(W) \supset A(X,U,Y,Z)}{[W, X \to Y, Z]C(W) \supset A(X,Y,Y,Z)}$$

as required. The functional composition rules are not relevant. The least fixpoint rule is as follows:

$$\frac{[X, U \to Y, Z]A[X,U,Y,Z] \quad (\forall XU)(\exists! Y)(\exists Z)A[X,U,Y,Z]}{[X \to Y, Z](A[X,Y,Y,Z] \wedge (\forall Y')((\exists Z')A[X,Y',Y',Z']) \supset Y \leq_d Y')}$$

We want to show the following:

$$\frac{[W, X, U \to Y, Z]C(W) \supset A[X,U,Y,Z] \quad (\forall XU)(\exists! Y)(\exists Z)A[X,U,Y,Z]}{[W, X \to Y, Z]C(W) \supset (A[X,Y,Y,Z] \wedge (\forall Y')((\exists Z')A[X,Y',Y',Z']) \supset Y \leq_d Y')}$$

Now, from the least fixpoint rule, substituting $C(W) \supset A[X,U,Y,Z]$ everywhere for $A[X,U,Y,Z]$ we obtain

$$\frac{[W, X, U \to Y, Z]C(W) \supset A[X,U,Y,Z] \quad (\forall WXU)(\exists! Y)(\exists Z)(C(W) \supset A[X,U,Y,Z])}{[W, X \to Y, Z]((C(W) \supset A[X,Y,Y,Z]) \wedge (\forall Y')((\exists Z')(C(W) \supset A[X,Y',Y',Z'])) \supset Y \leq_d Y')}$$

We have the following proof:

1.	$[W, X, U \to Y, Z]$	$C(W) \supset A[X,U,Y,Z]$ (assumption)
2.		$(\forall XU)(\exists! Y)(\exists Z)A[X,U,Y,Z]$ (assumption)
3.		$(\forall WXU)(\exists! Y)(\exists Z)(C(W) \supset A[X,U,Y,Z])$ (derivation in underlying logic)
4.	$[W, X \to Y, Z]$	$((C(W) \supset A[X,Y,Y,Z]) \wedge (\forall Y')((\exists Z')(C(W) \supset A[X,Y',Y',Z']))$ $\supset Y \leq_d Y')$ (least fixpoint rule)
5.	$[W, X \to Y, Z]$	$((C(W) \supset A[X,Y,Y,Z]) \wedge (\forall Y')((\exists Z')A[X,Y',Y',Z']) \supset Y \leq_d Y')$ (underlying logic rule)
6.	$[W, X \to Y, Z]$	$(C(W) \supset (A[X,Y,Y,Z]) \wedge (\forall Y')((\exists Z')A[X,Y',Y',Z']) \supset Y \leq_d Y')$ (underlying logic rule)

The underlying logic rule and all the remaining rules are fairly straightforward.

6.4 Applications to program generation

We give one application of the above deduction rules to program generation. Suppose we have done a derivation of the following form:

$$\frac{[\to Y_1]A_1(c) \ \ldots \ [\to Y_n]A_n(c)}{[X \to Y]A(c)}$$

where c is an uninterpreted constant symbol. This derivation essentially shows how a program satisfying A can be derived from programs satisfying A_i. This form of proof is convenient because it often corresponds to the calling structure of the procedures used. However, we may need to change this proof to a different form to prove that the specification A is always satisfied. By repeated application of the first deduction principle, we obtain

$$[Y_1 \ \ldots \ Y_n X \to Y](\wedge_i A_i(c) \supset A(c))$$

Since c is arbitrary, we can introduce a universal quantifier:

$$[Y_1 \ \ldots \ Y_n X \to Y](\forall x)(\wedge_i A_i(x) \supset A(x))$$

Now it may be possible to apply the recursion rule to make some of the Y_i identical to Y and then use mathematical induction to show $(\forall x)A(x)$. For this the variable X may be essential, since it can be used to record the argument on which a program is called. Such inductions typically make use of the fact that recursive calls use values of the parameters that are smaller in some well-founded ordering.

7 Extracting Program Constructing Functions from Proofs

We now show in a systematic way how to extract a program constructing function from a proof. Actually, a proof may be regarded as a constructive mapping from program constructing functions in the hypotheses, to a program-constructing function in the conclusion. For each rule, we give a description of this mapping and show that it is constructive. To indicate the mapping, we use the quantifier $[F : X \to Y]A[X, Y]$ which is interpreted as before as $(\forall X)A[X, F[X]]$. Here F is considered as a tuple of program-constructing functions. This construction also gives a more formal presentation of some of the reasoning in the proof of soundness of the logic.

Again we consider the following rules:

renaming rule	left elimination rule
permuting rule	right elimination rule
eliminating duplicates	introduction rule
recursion rule	right introduction rule
functional composition rules	conjunction rule
underlying logic rule	least fixpoint rule

For this discussion, we consider (F, G) to be a tuple of program constructing functions such that $(F, G)[X]$ is the concatenation of $F[X]$ and $G[X]$.

Corresponding to the renaming rule we have the rule

$$\frac{[F : X \to Y]A[X, Y]}{[F : U \to V]A[U, V]}$$

which is constructive since the F in the conclusion is obtained constructively (by the identity transformation) from the F in the hypothesis.

For the permutation rule we have

$$\frac{[F : X \to Y]A[X,Y]}{[F' : U \to V]A[U,V]}$$

where U and V are permutations of X and Y and where F' is defined so that $A[X, F[X]]$ is a variant of $A[U, F'[U]]$. This is done by defining F' as F with the arguments permuted and the elements of F permuted as necessary. This is still a constructive transformation, and is allowable since projections and composition are computable.

The rule eliminating duplicates does not change F since only the first occurrence of a variable in X or Y matters.

For the recursion rule we have

$$\frac{[(F_1, F_2) : X, U \to Y, Z]A[X, U, Y, Z]}{[(G, F_2) : X \to Y, Z]A[X, Y, Y, Z]}$$

where $F_1[X, U]$ and $F_2[X, U]$ correspond to Y and Z, respectively and where $F_1[X, G[X]] = G[X]$. Thus G is a fixpoint of F_1. Since we assume the fixpoint operator is constructive, this transformation from (F_1, F_2) to (G, F_2) is constructive.

The functional composition rules, where applicable, introduce a new program construction operation which is the composition of functional programs or the composition of functions and a predicate. This is assumed to be a constructive operation.

The underlying logic rule does not change F, and so the transformation is constructive.

For the left elimination rule, there are no program construction operations, and so the transformation is constructive. The hypothesis is of the form $[X, Y \to]A$, which means that F is empty.

We can express the right elimination rule in this way:

$$\frac{[(F, G) : X \to Y, Z]A}{[F : X \to Y](\exists Z)A}$$

This eliminates part of the given (F, G) tuple, and is therefore constructive.

The left introduction rule can be expressed as follows:

$$\frac{[F : X \to Y](\forall Z)A}{[G : Z, X \to Y]A}$$

where $G[Z, X]$ is $F[X]$. This is allowed because projection and composition are computable, and is constructive.

The right introduction rule is expressed as follows:

$$\frac{[F : X \to Y](\forall Z)A}{[(F, G) : X \to Y, Z]A}$$

where $G[X]$ is defined to be a fixed tuple Z of programs; this is allowable since we assume that there is at least one program of each sort, which is known constructively.

For the conjunction rule we have

$$\frac{[F : X \to Y]A \qquad [G : X \to Z]B}{[(F, G) : X \to Y, Z](A \wedge B)}$$

For the least fixpoint rule (when assumed) we have that the least fixpoint is constructible, so there is a constructible mapping giving the least fixpoint of a program generating function.

By repeatedly applying the above transformations we can obtain a program transforming function from a proof, if program transforming functions for the hypotheses are applied. Also, this can be done constructively. This provides a proof of the soundness of the logic as well as a way of extracting a program transforming function.

7.1 Specific programming languages

We generalize the construction of programs from proofs to permit the generation of programs in specific programming languages. This also permits us to give a more concrete description of the constructibility requirement on the logic. To do this, for each inference rule

$$\frac{[X_1 \rightarrow Y_1]A_1 \ \ldots\ [X_n \rightarrow Y_n]A_n}{[X \rightarrow Y]A}$$

we have to find a term $t(F_1, \ \ldots\ F_n)$ in the underlying logic such that the rule

$$\frac{[F_1 : X_1 \rightarrow Y_1]A_1 \ \ldots\ [F_n : X_n \rightarrow Y_n]A_n}{[t(F_1 \ \ldots\ F_n) : X \rightarrow Y]A}$$

is sound in the underlying logic, that is,

$[F_1 : X_1 \rightarrow Y_1]A_1 \wedge \ \ldots\ \wedge [F_n : X_n \rightarrow Y_n]A_n$ logically imply $[t(F_1 \ \ldots\ F_n) : X \rightarrow Y]A$. Also, these terms t must be constructible in two senses: They must have a computable operational semantics, and they must be constructible, that is, the terms in the conclusion of each inference rule must be constructible from the terms in the hypotheses. For this purpose, we might have some term like $fix(X, F(X,Y))$ for the recursion rule to represent a tuple G such that $F(G(Y),Y) = G(Y)$, and in this case, fix would have to be computable to produce the programs in the conclusion of the rule from programs in the hypotheses. Similarly, compositions and other operations used to construct programs in the inference rules must be computable. Furthermore, the programs so constructed must have a computable operational semantics. Using such rules repeatedly, we build up expressions representing program generating functions; these can be converted to programs when values for the unknown subprograms (or unknown program generating functions) are supplied. This transformation can be made more efficient by making use of derived rules of inference adapted to efficient constructions in the programming language. For example, certain recursions can be more efficiently translated into an iterative loop than a general recursion. Such recursions can be expressed in a derived rule of inference that captures the idea of a recursion that involves a counter (or whatever). By fashioning the proof so that such derived rules are used, one can obtain more efficient programs. This idea is also explored in a different program generation context by [1].

We illustrate the use of a derived rule for iteration to enable the generation of more efficient programs. Consider the following proof, where the underlying logic includes arithmetic and conditionals are computable:

1.	$[\to S]$	$P(S,0)$ (assumed)
2.	$[\to X]$	$(\forall n)(\forall w)(n \geq 0 \wedge P(w,n) \supset P(X(w), n+1))$ (assumed)
3.	$[S,X,Y \to Z]$	$(\forall n)(Z(n) = \ if \ n = 0 \ then \ S \ else \ X(Y(n-1)))$ (derived functional composition rule, using the computability of some functions)
4.	$[S,X \to Y]$	$(\forall n)(Y(n) = \ if \ n = 0 \ then \ S \ else \ X(Y(n-1)))$ (3, recursion rule, renaming)
5.	$[\to Y]$	$(\exists X,S)P(S,0) \wedge (\forall n)(\forall w)(n \geq 0 \wedge P(w,n) \supset P(X(w), n+1))$ $\wedge(Y(n) = \ if \ n = 0 \ then \ S \ else \ X(Y(n-1)))$ (1,2,4,composition rule)
6.	$[\to Y]$	$(\forall n)(n \geq 0 \supset P(Y(n), n))$ (5, underlying logic rule)

In this way we obtain the following derived rule of inference:

$$\frac{[\to S]P(S,0) \quad [\to X](\forall n)(\forall w)(n \geq 0 \wedge P(w,n) \supset P(X(w), n+1))}{[\to Y](\forall n)(n \geq 0 \supset P(Y(n), n))} \ (It_1)$$

The computed program would involve recursion, by the recursion rule in step 4. However, this computation can be done more efficiently by an iteration in many languages; the program $Y(n)$ can be expressed something like this:

```
w := S;
for i = 1 step 1 until n do W := X(w) od;
```

Therefore, by explicitly including this construction in the above derived inference rule, one could obtain a more efficient program. This would be a general optimization technique available when the underlying logic satisfied suitable additional assumptions. There are numerous opportunities for this kind of optimization, making use of special constructs in the target programming language.

7.2 Extracting concrete programs

In order to obtain an actual program, one has to have a proof in which none of the hypotheses assert the existence of program transforming functions. Also, the conclusion must be of the form $[\to Y]A$. Then the above constructions yield a program satisfying the specification A. However, we imagine that the system would often be used in a more abstract way, to reason about program constructing functions with generic assumptions. That is, if we can derive a formula A from formulae $A_1 \ldots A_n$, possibly asserting the existence of program constructing functions, and we can derive A_1 from formulae $B_1 \ldots B_m$, possibly also asserting the existence of program constructing functions, then we can derive A from $B_1 \ldots B_m A_2 \ldots A_n$. This involves putting together program constructing functions while still not constructing a concrete program. The ability to reason at this abstract level should increase the reusability and applicability of the proofs in this system. We note that in practice we will have a derivation of A' from $A_1 \ldots A_n$ and will need to show that A' implies A; this involves a kind of fitting together of programs. To do this, we may have to apply some kind of a proof homomorphism to the sub-proof, as explained below, to make the terminology consistent. We may also have to do some reasoning in the underlying logic. For example, A may be $[X \to Y]C$ and A' may be $[X \to Y]C'$. Then using the underlying logic rule it suffices to show that $\vdash (\forall X,Y)(C' \supset C)$. This amounts to showing that a given program satisfies a specification. Assuming that the underlying logic is classical, this step involves purely classical reasoning. We anticipate that a library of programs would be expressed in PL and derived with human assistance; then these programs could be combined for specific applications. This combining step would therefore be largely restricted to classical logic. We can regard the proof of A from $A_1 \ldots A_n$ as a parameterized program constructing function; when the program constructing functions for $A_1 \ldots A_n$ are supplied, we obtain a program constructing function A. This is somewhat like having generic programs for sorting, manipulating lists, and so on. Now, when we derive A_1 from $B_1 \ldots B_m$, we are to an extent instantiating the generic program constructing function for A_1, introducing in the process a number of other generic program constructing functions on which it depends. Note that this is done in a way that guarantees correctness; we can only instantiate the program constructing function for B_1 in a way that satisfies the specification B_1. Note the similarity of this approach to the use of abstract data structures.

7.3 Proof homomorphisms

It is possible to combine proofs in another way. For example, the choice of names for the functions and predicates in the logic is often arbitrary. One would like to be able to combine proofs using different naming conventions. Thus, one would like to translate a proof using one naming convention into a proof using another naming convention. Then the proofs could be combined. In general we can imagine a proof homomorphism H as a function mapping formulae in one underlying logic L_1 to another underlying logic L_2. We define $h([X \rightarrow Y]A)$ to be $[X \rightarrow Y]h(A)$ and therefore obtain a mapping between the programming logics PL_1 and PL_2 based on L_1 and L_2, respectively. Recall that $\vdash_{PL}$ indicates derivability in the logic PL, and similarly for logics PL_1 and PL_2. We say a mapping H is a proof homomorphism from PL_1 to PL_2 if it is a mapping from PL_1 to PL_2 and the following property is satisfied:

$$If\ A_1 \ \ldots\ A_n \vdash_{PL} 1A\ then\ h(A_1)\ \ldots\ h(A_n) \vdash_{PL_2} h(A).$$

We have the following easy result:

Proposition 7.1 *The mapping H is a proof homomorphism if the following conditions are satisfied:*

1. *All axioms of L_1 must map onto theorems of L_2.*
2. *If*

$$\frac{A_1 \ \ldots\ A_n}{A}$$

is an inference rule in L_1, then in L_2 we must have $h(A_1),\ \ldots,\ h(A_n) \vdash h(A)$.

Proof. By induction on the size of proofs.

□

In addition to simple mappings that involve changes in the names of functions and predicates, we have more interesting ones. For example, we can change $X < Y$ to $X > Y$ everywhere in many cases; using this mapping we can map a program to sort in increasing order, to a program to sort in decreasing order. Also, one might (for example) encode the integers as lambda calculus terms. Then, one could construct a proof homomorphism mapping results about the integers onto results about lambda calculus expressions. The use of such homomorphisms permits one to combine programs written with respect to different underlying logics.

In general, one can conveniently write general proofs that mention objects in the underlying logic (such as trees, terms, numbers, et cetera). When generating programs in some specific language, it may be necessary to find a more concrete representation of these objects; for example, we may need to represent trees or terms as lists in LISP. We can use a proof homomorphism for this purpose. Note that the original proof, mentioning abstract objects such as trees, is in this way more abstract than a proof in any concrete language, which must represent these objects in some way. This is one advantage of our representation of programs as proofs, over representing them in some specific language.

Another application of proof homomorphisms is to state variables. We may map an abstract proof onto one that corresponds to a program with an internal state; this may permit a more efficient use of data structures, for example. This could be formalized by adding an extra output to each function; this extra output would be the state resulting from computing the function. Our notion of proof homomorphisms is general enough to permit such a mapping that adds (or deletes) state information to (from) a program. Since our logic does not treat outputs of programs in any special way, such a mapping is possible. This also permits the combination of programs in different formalisms; we might have one program customized to list structures, and another that works on the level of abstract objects such as trees. We can combine them by first mapping the latter program onto a program in which trees are represented as lists. We might also be able to have a proof homomorphism mapping recursive constructions onto an explicit stack implementing recursion on a traditional von Neumann style architecture.

Used in this way, the programming logic PL has many of the advantages of algebraic specification methodologies [9] and other such algebraic specification methods: modularity, abstractness, reusability, and the guarantee of correctness with respect to a specification. However, this is obtained without a commitment to the initial algebra approach. Later we will also mention problems which rule out the direct use of the initial algebra approach in our method.

These proof homomorphisms also permit an abstract algorithm (a proof) to be realized in various programming languages. For this, we extend proof homomorphisms to statements of the form $[t : X \to Y]A$ by $h([t : X \to Y]A) = [h(t) : X \to Y]h(A)$. Thus the proof homomorphism can also map the terms t representing program generation functions. In this way we can obtain programs in different languages. We can verify the correctness of such proof homomorphisms in the same way as given above, but with attention paid to the $h(t)$ terms.

8 Abstract Logic Programming

We present an abstract approach that permits proofs to be converted into logic programs satisfying the specifications. This will include both term-rewriting systems, Horn-clause style logic programming, and functional programs as special cases. Then we give methods for showing that specific systems fit into the general framework. This framework automatically guarantees that the logic programs are constructible from a proof in PL. We write a logic program $L[X;Y]$ to indicate that X are input program variables and Y are output program variables. A logic program in this sense is just a formula of a special form in the underlying logic. It is intended that some special proof system will be used to derive input-output relations from logic programs. For example, term-rewriting can be used to derive equational consequences, or Prolog-style reasoning can be used to derive consequences of Horn clauses. Thus if X is a program variable, to compute $X(x_1 \ \ldots \ x_n)$ using L, we could derive a statement of the form $X(x_1 \ \ldots \ x_n) = y$ from L. Or, in the Prolog approach, we could derive $R_X(x_1, \ \ldots, \ x_n, y)$ for some suitable relation R_X.

Definition 8.1 *We say that a computable function F* satisfies *a formula $[X \to Y]A[X,Y]$ if $(\forall X)A(X, F(X))$. In this case we write $F \models [X \to Y]A[X,Y]$. We also say F satisfies $A[X,Y]$, and write $F \models A[X;Y]$.*

Definition 8.2 *We say that a logical expression* $L[X;Y]$ implements *a formula* $[X \to Y]A[X,Y]$ *if there is a computable function* F *that satisfies* $[X \to Y]L[X,Y]$ *and if* $(\forall XY)L[X,Y] \supset A[X,Y]$ *is valid in the underlying logic. We require that all the input variables* X *that occur in* L *must also occur in* A*, and all the output variables that occur in* A *must also occur in* L*. However, there may be input variables in* A *that do not appear in* L*, and there may be extra output variables in* L *that do not appear in* A*. Note that* $A[X,Y]$ *implements the formula* $[X \to Y]A[X,Y]$ *if this latter formula is derivable in* PL*.*

Definition 8.3 *A computable mapping* $G : L_1 \ \ldots \ L_n \to L$ *from logic programs to logic programs* realizes *an inference rule*

$$\frac{[X_1 \to Y_1]A_1 \ \ldots \ [X_n \to Y_n]A_n}{[X \to Y]A}$$

if for all $L_1 \ \ldots \ L_n$ *implementing the hypotheses,* $G(L_1 \ \ldots \ L_n)$ *implements the conclusion. Note that this allows customized realizations for derived inference rules. A computable mapping realizes a proof in the same way.*

Theorem 8.4 *If we have a proof in* PL *in which all inference rules are realized, then from logic programs implementing the hypotheses we can effectively obtain a logic program implementing the conclusion. That is, we can realize the whole proof.*

Proof. By a simple induction on proof size.

□

Corollary 8.5 *If we realize all the original inference rules in the system* PL*, then we can realize all proofs in* PL*.*

8.1 Abstract logic programming languages

We now consider abstract logic programming languages and their ability to realize proofs.

Definition 8.6 *A logic programming language is a set of formulas of the underlying logic, in which free program variables are specified as input and output variables.*

Definition 8.7 *We say that a logic programming language* LL *is* PL-adequate *if*

1. *Input and output variables can be renamed (subject to capture) (note that we need a subject to capture condition also for* PL*).*

2. *If* $L[X,U;Y,Z]$ *is a logic program then so is* $L[X;Y,Y,Z]$ *(or a logic program* $M[X;Y,Z]$ *such that* $M[X;Y,Z] \supset L[X;Y,Y,Z]$ *and such that for some constructible function* $h : PC \to PC$*, for all functions* F *and* G *in* PC*, if* $(F,G) \models L$ *then* $(H(F),G) \models M$*). That is, the input variables* U *have been replaced by output variables* Y*.*

3. *If* A *and* B *are logic programs with disjoint sets of output variables, then* $A \wedge B$ *is a logic program.*

4. *If* $t[F]$ *is a term involving functional program variables* F*, then there is a logic program* $L[G;F]$ *such that* $L[G;F] \models_{UL} (\forall x)G(x) = t[F](x)$*. Also, this logic program is constructible from the formula* $(\forall x)G(x) = t[F](x)$*. Here* x *can be a sequence of variables,* F *is a sequence of program variables, and* G *is a single program variable.*

5. *For all program sorts, there is a logic program* $L(;Y)$ *with one output variable* Y *and no input variables, implying* $Y = a$ *for some computable* a *of the appropriate sort.*

8.2 Annotating proofs with abstract logic programs

We now annotate the inference rules of PL to show how they can be realized by logic programs in a PL-adequate language LL. We call the resulting system PL_{prog}. Recall that in section 7.1 we used the notation $[F : X \rightarrow Y]A[X, Y]$ to indicate $A[X, F(X)]$, where F is in PC. Here we use the notation $[L : X \rightarrow Y]A[X, Y]$ to indicate that L is a logic program implementing A. This is not the same, since a logic program L could be satisfied by more than one function F. Now, it is possible for L to have output variables not explicitly mentioned in Y. First we consider the renaming rule, in which U and V are renamings of X and Y:

$$\frac{[L[X;Y] : X \rightarrow Y]A[X, Y]}{[L[U;V] : U \rightarrow V]A[U, V]}$$

Next we consider the rule eliminating duplicates; here U and V are X and Y, respectively, with duplicates removed:

$$\frac{[L[X;Y] : X \rightarrow Y]A[X, Y]}{[L[X;Y] : U \rightarrow V]A[U, V]}$$

Next we consider the permutation rule; here U and V are permutations of X and Y, respectively:

$$\frac{[L[X;Y] : X \rightarrow Y]A[X, Y]}{[L[X;Y] : U \rightarrow V]A[U, V]}$$

$$\frac{[L[X, U; Y, Z] : X, U \rightarrow Y, Z]A[X, U, Y, Z]}{[L[X; Y, Y, Z] : X \rightarrow Y, Z]A[X, Y, Y, Z]} \quad (\textit{Recursion rule})$$

where U and Y have the same sort.

$$\frac{[L[X, U; Y, Z] : X, U \rightarrow Y, Z]A[X, U, Y, Z]}{[M[X; Y, Z] : X \rightarrow Y, Z]A[X, Y, Y, Z]} \quad (\textit{Recursion rule})$$

(alternate version, where appropriate)
For functional programming:

(Functional composition rule 1)

$$[L[F, G] : F_1 \ldots F_m \rightarrow G](\forall z_1 \ldots z_n)G(z_1 \ldots, z_n) = T(F_1, \ldots, F_m, z_1, \ldots, z_n)$$

where $L[F, G]$ is a logic program such that $L[F, G] \models_{UL} (\forall z_1 \ldots z_n)G(z_1 \ldots, z_n) = T(F_1, \ldots, F_m, z_1, \ldots, z_n)$. (This must exist by point 4 in the definition of PL-adequate.)

(Underlying logic rule)

$$\frac{[L : X \rightarrow Y]A}{[L : X \rightarrow Y]B} \quad \textit{if} \ \vdash (\forall X, Y)(A \supset B) \ \textit{in the underlying logic L}$$

.

$$\frac{[L : X, Y \rightarrow]A}{[\textit{true} : Y \rightarrow](\forall X)A} \quad (\textit{Left elimination rule})$$

$$\frac{[L : X \rightarrow Y, Z]A}{[L : X \rightarrow Y](\exists Z)A} \quad (\textit{Right elimination rule})$$

$$\frac{[L : X \rightarrow Y](\forall Z)A}{[L : Z, X \rightarrow Y]A} \quad (Left\ introduction\ rule;\ Z, Y\ must\ be\ disjoint)$$

$$\frac{[L[X;Y] : X \rightarrow Y](\forall Z)A}{[L[X;Y] \wedge L'[;Z] : X \rightarrow Y, Z]A} \quad (Right\ introduction\ rule;\ Z, X\ must\ be\ disjoint)$$

Here $L'[;Z]$ is a conjunction of $L'[;Z_i]$ for Z_i in Z, whose existence is required by point 5 in the definition of PL-adequate.

$$\frac{[L_1 : X \rightarrow Y]A \quad [L_2 : X \rightarrow Z]B}{[L_1 \wedge L_2 : X \rightarrow Y, Z](A \wedge B)} \quad (Conjunction\ rule;\ if\ no\ captures\ and\ Y, Z\ are\ disjoint)$$

Theorem 8.8 *If LL is PL-adequate then all proofs in PL can be realized by logic programs.*

Proof. We show that the above annotated inference rules in PL_{prog} are all realized by the indicated logic programs. We first note that the transformations to the required logic programs in the conclusions are all computable. It remains to show that if the indicated logic programs implement the hypotheses, then the indicated logic program in the conclusion implements the conclusion. This is mostly quite straightforward. For example, for the recursion rule, we note that if $(\forall XUYZ)(L[X,U;Y,Z] \supset A[X,U,Y,Z])$ then $(\forall XYZ)(L[X;Y,Y,Z] \supset A[X,Y,Y,Z])$. The alternate version of the recursion rule is similar, since $M[X;Y,Z] \supset L[X;Y,Y,Z]$. For now we ignore the second functional composition rule, assuming that a predicate is regarded as a Boolean function. Or else we can extend 4. to predicates too. We ignore the least fixpoint rule for the time being. The underlying logic rule requires no change in the logic program. Perhaps it would be appropriate to give this argument in more detail. Suppose we derive $[X \rightarrow Y]B$ from $[X \rightarrow Y]A$ in PL. Let L be a logic program implementing $[X \rightarrow Y]A$. Then we know that there is a computable function satisfying $[X \rightarrow Y]L$. Also, $(\forall XY)(L \supset A)$. Since $A \vdash B$ in the underlying logic, $(\forall XY)(L \supset B)$ also. Therefore L implements B.

We now consider the left elimination rule. Suppose that $[X, Y \rightarrow]A[X,Y]$ and we have a logic program L implementing this rule. Then there is a computable function F such that $(\forall X,Y)A[X,Y,F(X,Y)]$, but since A has no output variables, we have $(\forall XY)A[X,Y]$. Therefore we can take L to be just true, without any input or output variables, and L implements $[X,Y \rightarrow]A[X,Y]$. It is easy to see that L also implements $[Y \rightarrow](\forall X)A$. The right elimination rule is handled as follows: Suppose $L[X;Y,Z]$ implements $[X \rightarrow Y,Z]A$. Then $(\forall XYZ)(L[X;Y,Z] \supset A)$. Therefore $(\forall XYW)(L[X;Y,W] \supset (\exists Z)A)$. This is all that is required; we are allowed to have the extra W output variables in L. The left introduction rule is no problem; A receives new input variables that do not appear in L. The conjunction rule is dealt with by noting that the conjunction of two logic programs is a logic program, and that the output variables of the hypotheses are distinct. The right introduction rule introduces new output variables in A. For this, we need to introduce new output variables in L. By point 5, we can find a logic program with a new output variable. By point 3, we can take the conjunction of this logic program with another, adding a new output variable. By repeating this process, an arbitrary number of new output variables can be added. This is enough for the right introduction rule, since the hypothesis must be true for all Z.

□

What is slightly surprising about this theorem is that no quantifiers are required. Although the inference rules of PL may add or remove universal and existential quantifiers, these can be simulated by logic programs without any quantifiers at the outer level.

8.3 Observable behavior of logic programs

We now consider the computational aspect of logic programming languages. We assume that there is an *interface language* IL associated with LL and a (sound) inference operation $\vdash_{LL}$ for deriving IL assertions from LL programs. For example, for term-rewriting systems, the assertions in IL might be of the form $F(s_1 \ldots s_n) = t$ where $s_1, \ldots, s_n$, and t are constructor terms and F is a defined function. Thus we evaluate F on inputs $s_1 \ldots s_n$ and obtain the output t. Here the inference rules $\vdash_{LL}$ could be term-rewriting. For Prolog-style Horn-clause logic programming, the IL-assertions might be of the form $P(s_1 \ldots s_n)$ where P is a predicate and $s_1 \ldots s_n$ are terms, indicating that the terms s_i satisfy P. This could be returned by a logic program as an answer to a query of the form :- $P(r_1 \ldots r_n)$ for terms r_i. The inference rules here could be SLD-resolution for deriving such assertions $P(s_1 \ldots s_n)$. The interface language corresponds to the input-output behavior of LL programs that is visible to the user. We require that the relation $\vdash_{LL}$ be (partially) computable. That is to say, given A, one should be able to enumerate the formulas B such that $A \vdash_{LL} B$. This corresponds to the fact that logic programs should have a computable operational semantics.

Definition 8.9 *Suppose $L(X;Y)$ is a logic program and IL is an interface for it. Then the (IL-)*interface *of L is the set of $B(X;Y)$ in IL such that $L(X;Y) \vdash_{LL} B(X;Y)$. Here the X and Y are considered as free variables, so we really have that for any disjoint X and Y, $L(X;Y) \vdash_{LL} B(X;Y)$. The interface of L is the observable part of L, as far as direct input-output behavior is concerned.*

We now say something about the correctness of the derived logic programs. We often abbreviate $L[;Y]$ and $B(;Y)$ by $L(Y)$ and $B(Y)$.

Theorem 8.10 *Suppose $L[;Y]$ implements $[\rightarrow Y]A(Y)$. Then there is a computable function F such that $A(F)$ and such that if $B(Y)$ is in the interface of $L[;Y]$ then $B(F)$. That is, the interface of L can be extended to a computable function satisfying the specification A.*

Proof. Since L implements $[\rightarrow Y]A(Y)$, there is a computable function F that satisfies $[\rightarrow Y]L[;Y]$. In this case this implies $L[;F]$. Since $(\forall Y)(L[;Y] \supset A(Y))$, we have also $A(F)$. Now, if $B(Y)$ is in the interface of $L(;Y)$, then $L[;Y] \vdash_{LL} B(Y)$ hence $L[;F] \vdash_{LL} B(F)$. Thus $B(F)$ for all B in the interface, and F satisfies A.

□

In particular cases, we may be able to derive many such B, and then we know that these are correctly computed and can be extended to a computable function as stated above. It remains to say more about how much of F can be computed in general. The above theorem does not rule out the possibility that the interface of L can be empty, which for functional programs might mean that for no inputs $s_1 \ldots s_n$ is the output $t = F(s_1 \ldots s_n)$ computable. We will deriving conditions under which the deduction $\vdash_{LL}$ is "complete," in a sense. The idea is to show that there is a computable function F satisfying L such that for all B in IL, $B(F)$ iff $L(Y) \vdash_{LL} B(Y)$. Thus the interface of F is the intersection of the interfaces of all functions satisfying L, so F is a kind of "initial" object. This function F must also satisfy A, because $L \supset A$, and so if $A(Y) \supset B(Y)$ for B in IL, we have that $L(Y) \supset B(Y)$, hence $B(F)$, so by definition of F, B is in the interface of L. Thus all $B(Y)$ in IL for which $A(Y) \supset B(Y)$, are in the interface of L, and hence visible to the user. The difficulty in general is nontermination; it is not easy to compute nonterminating elements, and the inputs or results of a computation may be nonterminating. These are considered as part of a computable function, but do not appear in the interface.

Definition 8.11 *A logic programming language LL is* extensible *for interface IL if for all logic programs L in LL there exists a computable function F such that $L(F)$ and such that for all B in IL, $\models_{UL} B(F)$ iff $L(X) \vdash_{LL} B(X)$.*

Theorem 8.12 *If LL is extensible for interface IL, L is in LL and L implements A, and $(\forall Y)(A(Y) \supset B(Y))$, then B is in the interface of L.*

Proof. Since LL is extensible for IL, there is a computable F such that $L(F)$ and such that for all B in IL, $B(F)$ iff $L(X) \vdash_{LL} B(X)$. Since $L(F)$ and $L \supset A$, $A(F)$. Since $(\forall Y)(A(Y) \supset B(Y))$, $B(F)$. Therefore, by definition of F, $L(X) \vdash_{LL} B(X)$.

□

We now want to give conditions under which L cannot derive "too much." The preceding theorem still allows L to have an interface possibly larger than that needed for F.

Definition 8.13 *Suppose L_1 and L_2 are logic programs in LL. Let IL be an interface language for LL. Then L_1 and L_2 are* interface-equivalent *(for IL) if for all assertions B in IL, $L_1 \vdash B$ iff $L_2 \vdash B$.*

Definition 8.14 *A specification $A(F)$ is* complete *with respect to an interface language IL iff $A(F_1) \wedge A(F_2)$ implies that F_1 and F_2 are interface-equivalent for IL.*

Theorem 8.15 *Suppose L implements A and A is complete with respect to interface language IL. Suppose $B(Y)$ is in the interface of L. Then $(\forall Y)(A(Y) \supset B(Y))$.*

Proof. We know there is a computable function F such that $L(F)$. Since B is in the interface of L, $B(F)$. Since $L \supset A$, $A(F)$. Suppose $A(Y)$; then Y is interface-equivalent to F. Therefore $B(Y)$ also.

□

Corollary 8.16 *Suppose LL is extensible for interface IL, L is in LL, L implements A, and A is complete with respect to IL. Then B is in the interface of L iff $(\forall Y)(A(Y) \supset B(Y))$.*

Proof. By a combination of the two preceding theorems.

□

Completeness cannot be guaranteed since it depends on A, over which we have no control. The hardest part of the above is proving extensibility. For languages having a denotational semantics based on complete partially ordered sets and least fixpoints, extensibility is typically fairly straightforward. For such languages, the recursion rule corresponds to a least fixpoint operation, and for each logic program LL derived there will be a least computable function F satisfying LL. It follows that assertions of the form $F(s_1 \ \ldots \ s_n) = t$, where s_i and t are maximal in the domain ordering, will be true for all F satisfying LL. Maximal elements are "defined," that is, they contain no occurrences of $\perp$, typically. Furthermore, if t is "finite" (for example, not an infinite list), then such assertions can be derived by a finite computation (or derivation). Therefore, we may take the interface to be the set of such assertions $F(s_1 \ \ldots \ s_n) = t$ where the s_i and t are defined and finite; this guarantees extensibility. However, for some programming formalisms, the existence of a denotational semantics is not straightforward; this includes term-rewriting systems (for whatever reason) and logic programs (because of the nondeterminism).

Another approach is to use initial algebras [9]; it is known that any set of equations has an initial algebra. One problem is that this algebra may be only partially computable; the equality may not be decidable. Another problem is that it may involve functions that do not possess fixpoints, as required by our definition of a programming structure. For example, consider the empty set of equations with the constructors 0 and s (successor). The initial algebra is the natural numbers. However, s must have a fixpoint. We usually require for constructors that $s(x) = s(y)$ iff $x = y$. This implies that the fixpoint of s must be the infinite term $s(s(s\ \ldots))$. In addition, we typically need $\perp$ to represent undefined values. Thus the semantics will need to contain infinite terms with occurrences of "bottom." In this way, the initial algebra approach becomes considerably more complicated, and not much different than the denotational semantic approach above. We still may get extensibility this way if the underlying logic is too weak to distinguish the initial algebra from computable functions, however.

8.4 Evaluation strategies and underlying logics

We may specify a number of different relations $\vdash_{LL}$ for a given logic programming language LL, and these have implications for the underlying logic. These different relations $\vdash_{LL}$ correspond to different methods for deriving consequences of a logic program L. These may correspond to different evaluation strategies (for example, lazy or eager evaluation), that is, different operational semantics. A strategy that permits more consequences of L to be derived, restricts the semantics more, since models of UL must satisfy all these consequences. A strategy that permits fewer consequences to be derived allows more flexibility in the choice of semantics, and sometimes permits a simpler semantics. This has a corresponding effect on the underlying logic, since it should be sound with respect to the semantics. In particular, the more models there are, the fewer inferences are sound in the underlying logic, and the more restrictive the underlying logic rule is. The fewer models there are, the more axioms and inferences can be used in the underlying logic rule. We will illustrate these interrelationships below.

8.5 Generating term-rewriting programs

We extend the inference rules to generate assertions of the form $[R(X;Y) : X \rightarrow Y]A[X,Y]$ where $R(X;Y)$ is a term-rewriting system mentioning the variables X and Y as function symbols. For a survey of term-rewriting systems, see Dershowitz and Jouannaud [13] or [30]. To formalize the generation of such systems, we define a *term-rewriting program* $R(X;Y)$ as a set (conjunction) of equations $\{r_1 = s_1\ \ldots\ r_n = s_n\}$ in which the orientation of the equations matters, that is, which term is on the left and which is on the right. These equations may contain function symbols from the underlying logic as well as the variables X and Y as function symbols. These equations are viewed computationally as the term-rewriting system $R_{\rightarrow}(X;Y) = \{r_1 \rightarrow s_1\ \ldots\ r_n \rightarrow s_n\}$. We require that these rules be orthogonal, that is, left linear and non-overlapping. We cannot in general require termination, since many reasonable programs do not correspond to terminating term-rewriting systems. However, left linearity is reasonable, since the left-hand sides correspond roughly to procedure calls, and the formal parameters of a procedure definition are typically distinct. A subset of the function symbols from the underlying logic are called constructor terms. Also, the left-hand sides r_i are all of the form $F(u_1\ \ldots\ u_n)$ where F is an output variable or a non-constructor function symbol, and the u_i are constructor terms. This is called the *constructor discipline*. The right-hand sides s_i may contain input variables, output variables, and arbitrary function symbols from the underlying logic. We consider LL_{tr} as the logic programming language specified in this way, that is, the set of term-rewriting programs. The inference rule is replacement of equals by equals left to right, which is computable, and the interface language is the set of equations of the form $F(s_1 \ldots s_n) = t$ where $s_1 \ldots s_n$ and t are constructor terms and F is a program variable or a function from the underlying logic.

Theorem 8.17 *The language LL_{tr} is PL-adequate.*

Proof. The points 1 and 2 are immediate. For 3, we have non-overlapping because the output variables of the hypotheses L_1 and L_2 are distinct. For point 4, we need to add a rule $F(x) = t[F, G](x)$, which is allowed because F is an output program variable. For point 5, we can have the system $Y = a$.

□

We note that this implies, that if in a PL-proof, the assumptions are implemented by LL_{tr} programs, then we can effectively obtain an LL_{tr} program implementing the conclusion. This requires, among other things, that the term-rewriting systems implementing the assumptions be orthogonal, and then guarantees that the system implementing the conclusion will be orthogonal. These systems implementing the assumptions will often compute functions in the underlying logic (such as addition, multiplication, et cetera). At the lowest level, we can assume that the constructors and destructors are computable, and also a conditional function that tests the top-level constructor of a term. It is easily verified that these functions are in fact computable for reasonable representations of terms; however, nontermination has to be handled propery, as indicated below. Then other functions can be defined in terms of these basic functions. We also note that all derived LL_{tr} programs will have at most one rule of the form $Y_i(u_1 \ldots u_n) = s$ for each output variable Y_i, and for such a rule, $u_1 \ldots u_n$ will be distinct variables, assuming that the systems implementing the hypotheses also have this property. The reason is that such rules can only be introduced by the functional composition rule, and all transformations preserve this property. There may be two or more rules $r_i = s_i$ for which all r_i have the same top-level symbol F from the underlying logic, however.

Theorem 8.18 *For $R(X;Y)$ in LL_{tr}, $R_{\rightarrow}(X;Y)$ is confluent. Also, parallel- outermost rewriting will compute a normal form, if one exists. Furthermore, if F is an output variable and $u_1 \ldots u_n$ are constructor terms and t is a constructor term and the equation $F(u_1 \ldots u_n) = t$ is a logical consequence of $R(X;Y)$ using only equality reasoning, then t is the R-normal form of $F(u_1 \ldots u_n)$, and t can be effectively computed from $F(u_1 \ldots u_n)$.*

Proof. $R_{\rightarrow}(X;Y)$ is confluent because it is orthogonal. For orthogonal systems, parallel-outermost rewriting computes normal forms, if they exist. For the last part, the Church-Rosser property of confluent systems guarantees that t will be the normal form of $F(u_1 \ldots u_n)$, since t is irreducible. Also, t can be effectively computed, since parallel-outermost rewriting is effective.

□

This shows that we can derive a certain portion of the input-output relation of the function F, in fact, its entire interface. However, this is still unsatisfactory, because it is related not to the final specification of F, but rather to the logic program R, which is constructed without the user's control or possibly even without his or her knowledge. We would rather relate the computability of F to the derived specification of F. That is, if we prove $[R : X \rightarrow Y]A$, we would rather relate the computability of Y to the specification A instead of to R. We know that

$R \supset A$; if in fact $A \supset R$ also, then A and R are equivalent, and all equational consequences of R are also consequences of A. This implies that all equational consequences of R of the form $F(u_1 \ldots u_n) = t$ as above, are also consequences of A. Most of the rules of PL_{prog} actually preserve equivalence, but some of them do not. This corresponds to the fact that the program R may give more information than the specification. Another case of interest is when all the reasoning in the proof of $[R : X \to Y]A$ is equational; then any equational consequence of A is also an equational consequence of R, and is therefore derivable from R by rewriting. However, it still may be possible to derive extra consequences from R that are not derivable from A. If the specification of A is complete, then we know that there is essentially only one computable function (or sequence of functions) satisfying A, so R and A are equivalent (with respect to the interface), and elements of the interface of R are consequences of A. However, it is possible that the underlying logic permits many interesting consequences of R to be derived by non-equality reasoning, which we may miss.

To solve this, we use the theorems about completeness and extensibility given earlier. For this we need to show how to extend the interface of a term-rewriting system to a function satisfying the equations in the system. A problem is that for some $u_1 \ldots u_n$, $F(u_1 \ldots u_n)$ may have no normal form. That is, the computation of $F(u_1 \ldots u_n)$ may not terminate. We need to find a function F' satisfying the interface, which means that $F'(u_1 \ldots u_n)$ has to have a value and these values need to be chosen to satisfy R. We can partially solve this by adding a $\perp$ element and the equation $x \neq \perp \supset \perp <_d x$. However, just representing all such $F(u_1 \ldots u_n)$ by $\perp$ won't do, because sometimes they can behave differently in some contexts. For example, $F(u_1 \ldots u_n)$ may compute an infinite list of integers, and it may be possible to extract the third element of this list. This infinite list program is, incidentally, an example of a nonterminating program that our formalism can handle. A program to generate such an infinite list can be implemented for example by the term-rewriting system $list(n) \to cons(n, list(s(n)))$, which could easily be generated in our system. Two different such infinite lists (say, $list(0)$ and $list(1)$) may have different third elements, and thus cannot be both equal to "bottom," even though they both fail to terminate. Another solution is just to consider terminating systems; this doesn't suffice because many natural definitions (for example of the factorial function) correspond to non-terminating systems.

One solution is to consider restricted evaluation strategies, such as innermost rewriting ; this amounts to ensuring strictness. Let $\mathcal{T}(\mathcal{F}, \mathcal{X})$ be the set of terms over a set $\mathcal{F}$ of function symbols and a set $\mathcal{X}$ of variables. Then we consider the flat domain $\mathcal{T}(\mathcal{F}, \mathcal{X}) \cup \{\perp\}$, and the programs are then continuous (strict) functions from this domain to itself. The program construction functions are then continuous functions from a tuple of such program domains to a program domain. Strictness of programs requires, for example, that if-then-else always evaluates all of its arguments, leading to undesired nontermination. This means also that the destructors arg_i no longer satisfy the equation $arg_i(f(x_1, \ldots, x_n)) = x_i$, since if some x_j is $\perp$ then $f(x_1, \ldots, x_n) = \perp$; instead we can only say that $f(x_1, \ldots, x_n) \neq \perp \supset arg_i(f(x_1, \ldots, x_n)) = x_i$. Similary, the test $top\_f(t)$ whether the top level function symbol of t is f no longer satisfies $top\_f(f(\ \ldots\ )) = true$; instead we have $f(\ \ldots\ ) \neq \perp \supset top\_f(f(\ \ldots\ )) = true$. Also, we can now only say that $f \neq g \wedge g(\ldots) \neq \perp \supset top\_f(g(\ldots)) = false$. As mentioned above, this corresponds to a restriction on the derivation relation $\vdash_{LL}$ and allows more models; we take flat domains and require all functions to be strict. This has the effect of allowing more rules in the underlying logic (for example, $f(\ \ldots,\ \perp,\ \ldots\ ) = \perp$). This works, but results in assigning too many terms a value of "bottom" that have defined values when non-strict computation is used.

Yet another idea is to use a reduction strategy which evaluates $f(u_1 \ \ldots \ u_n)$ carefully; only some of the arguments are chosen for evaluation (depending on f and on the results of previous evaluations), but whenever an argument is selected, it is evaluated all the way to normal form. This makes all nonterminating terms equivalent, which corresponds to a flat domain but with non-strict functions allowed. Thus the program domains would contain continuous functions from tuples of $\mathcal{T}(\mathcal{F}, \mathcal{X}) \cup \{\perp\}$ to itself, and the program construction functions would contain continouus functions from tuples of program domains to a program domain. Again, this corresponds to a different derivability relation $\vdash_{LL}$ and influences the semantics. Since the domains are flat, we can use the rule $x < y \supset x = \perp$ in the underlying logic, for example. However, the function programs cannot be assumed to be strict. This approach permits a reasonable number of functions (including those involving conditionals) to be computed without undesirable nontermination. For this, we evaluate if-then-else in a reasonable way, with the first argument evaluated to normal form, then reducing the whole to one of the two remaining arguments. Thus we have the equations $if(true, y, z) = y$ (even if $z = \perp$) and $if(false, y, z) = z$ (even if $y = \perp$). For other functions with specialized evaluation strategy, corresponding equations can be added to the underlying logic. In addition, the domain structure is fairly simple. However, some of the flexibility of lazy evaluation is lost.

A further solution is to consider the values in a non-flat domain, and construct the values of non-terminating computations by (non-strict) least fixpoints. Maybe there are other solutions too. This solution works for term-rewriting systems of the kind we are considering; the semantics is given in terms of infinite terms containing occurrences of "bottom" and ordered so that $s >_d t$ if t is obtained from s by replacing some set of subterms by $\perp$. Thus $f(a, b, c) >_d f(a, \perp, c)$. For this, we can add the following equations to the underlying logic, for all constructors f:

$$\begin{gathered}
arg_1(f(x_1, \ \ldots, \ x_n)) = x_1 \\
arg_2(f(x_1, \ \ldots, \ x_n)) = x_2 \\
\ldots \\
arg_n(f(x_1, \ \ldots, \ x_n)) = x_n \\
f(x_1 \ \ldots \ x_n) = f(y_1 \ \ldots \ y_n) \supset x_1 = y_1 \wedge \ \ldots \ \wedge x_n = y_n \\
f(\ \ldots \ \perp \ \ldots \ ) \neq \perp \\
f \neq g \supset f(\ \ldots \ ) \neq g(\ \ldots \ ) \\
top_f(f(\ \ldots \ )) = true \\
f \neq g \supset top_f(g(\ \ldots \ )) = false \\
x >_d y \supset f(\ \ldots \ x \ \ldots \ ) >_d f(\ \ldots \ y \ \ldots \ )
\end{gathered}$$

Given a term s, we can define $bot(s)$ to be s with all maximal subterms $f(t_1, \ \ldots, \ t_n)$ for all non-constructors f, replaced by $\perp$. Then one can show that $\{bot(t) : s \rightarrow^* t\}$ has a least upper bound, a possibly infinite term which may contain constructors and occurrences of $\perp$, which can be taken as the semantics of s. The reason that this least upper bound must exist is that confluence implies that if some t has a constructor in a given position, then no other t can have a different constructor there. In fact, to make the computation easier, we can use the least upper bound of $\{bot(t) : s \rightarrow^*_{po} t\}$ where $\rightarrow_{po}$ indicates parallel outermost reduction. This is computable, as required, since if there is a constructor at a given position in this least upper bound, eventually a term will be generated having that constructor there, and if there is an occurrence of $\perp$ somewhere, then the search for this constructor will not terminate (or will terminate with a term having a non-constructor symbol at that position). Also, we can show that the semantics of $f(t_1 \ \ldots \ t_n)$ is a function of the semantics of $t_1 \ \ldots \ t_n$; this is true because the constructor condition implies that the only parts of the t_i that can be "seen" by the computation of F are the constructors that eventually emerge at the top. We therefore need to define the domain as the set of such infinite terms, and show how $f(t_1 \ \ldots \ t_n)$ is defined when t_i are also infinite terms; this can be done by assuming f is continuous. As before, this gives us a domain for the programs; they are continuous functions from tuples of such infinite tree domains, to an infinite tree domain. Then the program construction functions are continuous functions from tuples of program domains to a program domain.

This influences the underlying logic; for example, we can now assume that $f(x) \neq x$ for a constructor f and all x. This axiom is not satisfied by flat domains. In this way, we obtain that the computed function F' satisfies the logic program R. Then we have that all logical consequences of A of the form $F(u_1 \ldots u_n) = t$, for constructor terms u_i and t, are true of F' too, that is, $F'(u_1 \ldots u_n) = t$, so this can be computed by term-rewriting using parallel outermost rewriting. For this it is necessary to assume that the models of the underlying logic are rich enough to express such infinite tree semantics for R. Seen another way, we have to insure that the axioms of the underlying logic are satisfied by such a semantics. We note that if the wrong underlying logic is chosen, then unsound consequences can be derived; for example, if we assume that the domain is flat, then all but finitely many elements of the sequence $\perp$, $cons(0, \perp)$, $cons(0, cons(1, \perp))$, $\ldots$ must be equal. Using $cdr(cons(x, y)) = y$ and $car(cons(x, y)) = x$, we have that $\perp = cons(i, cons(i + 1, \ldots))$, so $car(\perp) = i$, and this must hold for infinitely many i (an obvious impossibility). If A is complete, then we know that parallel outermost rewriting will only compute consequences of A. Thus we can completely characterize which equations of the form $F(u_1 \ldots u_n) = t$ can be computed by R with parallel outermost rewriting (exactly the underlying logical consequences of A).

8.6 Horn clause logic programming

We now give an example of a logic programming language based on Horn clauses. A *Horn clause logic program* is a conjunction of Horn clauses L :- $L_1 \ldots L_n$, where each L and L_i are positive literals of the form $eval(s, t)$ for terms s and t, or of the form $s = t$. Input and output program variables may appear in s and t. The interface is statements of the form $f(s_1 \ldots s_n) = t$ for constructor terms s_i and t. The inference rule $\vdash_{LL}$ is SLD-resolution. Examples of clauses in this language are

eval(plus(s(x),y),s(z)) :- eval(plus(x,y),z).
eval(plus(0,y),y).

The intuition is that $eval(s, t)$ is evaluating a term s to the value t. The literals $s = t$ are not used in the computation, but rather to insure PL-adequacy. We need to show PL-adequacy. For this, we need to show that if $t[F, G]$ is a term involving functional program variables F, then there is a logic program $L[G; F]$ such that $L[G; F] \models_{UL} (\forall x)F(x) = t[F, G](x)$. We can take the logic program

eval(F(x),z) :- eval(t[F,G](x),z).
x = y :- eval(x,y).
F(x) = t[F,G](x).

The first clause implements outermost rewriting to evaluate $F(x)$. The last clause does not enter the computation at all. The middle clause answers queries of the form $F(s_1 \ldots s_n) = t$, and therefore serves a computational purpose in generating the interface. We also need to show that for all progam sorts, there is a logic program $L(; Y)$ with one output variable Y and no input variables, implying $Y = a$ for some computable a of the appropriate sort. For this we can use the following logic program:

```
eval(Y,z) :- eval(a,z).
eval(a,a).
```

This is enough for PL-adequacy. We can implement a conditional as follows, avoiding unnecessary evaluations, and permitting the computation of non-strict functions:

```
eval(if(x,y,z),w) :- eval(x,xv), eval(if2(xv,y,z),w).
eval(if2(true,y,z),w) :- eval(y,w).
eval(if2(false,y,z),w) :- eval(z,w).
```

For constructors F and constructor constants c we would have

```
eval(f(x1 ... xn),f(y1 ... yn)) :- eval(x1,y1), ... , eval(xn,yn).
eval(c,c).
```

We can also implement destructors as follows:

```
eval(arg1(f(x1 ... xn),x1)).
eval(arg2(f(x1 ... xn),x2)).
...
```

Finally, we can test for the top level constructor of a term as follows:

```
eval(top_f(f(x1 ... xn), true)).
eval(top_f(g( ... )), false).
...
```

This is enough to build up other computable functions. The clauses as given here evaluate a term by repeatedly rewriting at the top level until a constructor term appears, and then recursively evaluating the arguments, or else a non-constructor symbol from the underlying logic appears at the top level, for which a customized program does the evaluation. Since Horn clause style programming is very expressive, it should not be hard to see how arbitrary programs can be written. What our approach adds is an automatic way to generate such logic programs from a proof.

We note that the generation of Horn clause programs as given here is weaker than the term-rewriting translation with parallel outermost rewriting. For example, to evaluate $or(s,t)$, the Horn clause translation would have to pick an order in which to evaluate s and t. It might choose s first, which nonterminates, while t may evaluate to true. The term-rewriting approach could evaluate both s and t in parallel. We can give a direct implementation of disjunction ("or") as follows:

```
eval(or(x,y),true) :- eval(x,true).
eval(or(x,y),true) :- eval(y,true).
eval(or(x,y),false) :- eval(x,false), eval(y,false).
```

This can be made to evaluate both x and y in parallel, if some complete theorem proving strategy is applied (such as breadth-first search). Similarly, the other logical connectives can be implemented; for example, we have

```
eval(not(x),true) :- eval(x,false).
eval(not(x),false) :- eval(x,true).
```

As for extensibility, it is known [22] that the deductive and least model semantics of Horn clauses agree. This is a partial answer to the question. However, this does not completely settle the issue, because we are looking for a semantics of the function symbols that satisfy the logic program (including the given equations). The typical logic programming semantics interprets the function symbols syntactically, that is, as constructors. However, as shown above, with the existence of fixpoints this requires that the model contain infinite terms, which are not in the standard least model of a set of Horn clauses. Therefore it is necessary to define the semantics in terms of infinite trees; another possibility is to use a flat domain and give up the identity $f(x) = f(y) \equiv x = y$ or the rule $f(x) \neq x$ for constructors f. Of course, this restricts the evaluation strategies that can be used to compute such functions. Another possibility is to say that the constructors are not computable; this formally solves the problem, but is intuitively unsatisfying and also makes extensions of the language difficult. As for evaluation strategies, note that depth-first search as usually done in Prolog is incomplete, so a better (but slower) search method is needed in general to guarantee that all formulae in the interface can be generated. The translation as defined above, however (except for the implementation of "or"), will work for depth-first search with any ordering of the clauses and literals because of the absence of nondeterminism in the resulting Horn clause program. That is, depth-first search gives an operational semantics permitting the same interface to be generated as by a complete breadth-first search.

In general, for Prolog it is convenient to consider predicates as mappings from tuples of ground terms to {true,false}, with false identified with $\perp$. This turns out to be reasonable, because it is possible to enumerate the true literals $P(t_1, \ldots, t_n)$ in the minimal model of a set of Horn clauses. If such a literal is true, that can eventually be detected, but if it is false, the search may not terminate, which corresponds to $\perp$. Then the predicates can be seen as continuous functions from the set of ground terms (with $\perp$) to $\{\perp, \text{true}\}$, and the program construction functions can be taken as continuous functions from tuples of such program domains, to a program domain. This does not treat the constructors as computable; this can be remedied by interpreting them as strict functions over a flat domain of ground terms. One shortcoming of this approach is that it does not distinguish nontermination from finite failure; another is that it does not explicitly consider the answer substitutions returned for a non-ground query. But the purpose is just to sketch how our general framework can be applied to Horn clause logic programming, and not to give an exhaustive account.

One interesting feature of the logic programming approach is that it permits nondeterminism. We can allow a "choice" operator where $choice(x, y)$ can evaluate to either x or y. This can be implemented in Prolog as follows:

```
eval(choice(x,y),z) :- eval(x,z).
eval(choice(x,y),z) :- eval(y,z).
```

Of course, to show extensibility, it now becomes necessary to use some kind of a nondeterministic denotational semantics.

This approach does not fully exploit the power of Prolog, since the programs derived are of a very special form. It would be interesting to find other PL-adequate logic programming languages based on Horn clauses. One possibility would be to look at methods of compiling concept description languages such as the KL-ONE system of [7] into Horn clauses.

8.7 von Neumann machines

The Horn clause implementation can be made to look a lot more like a von Neumann language. The "logic programs" in this case are sequences of procedure definitions, in which the program variables appear as the names of the procedures. A formal semantics would interpret these programs as state transformations. For this, we assume that terms on right-hand sides of assignment statements are evaluated, but the actual parameters to procedures are not evaluated and that the statement "return(y)" does not evaluate y (in order to make the structure of the computation more explicit). We mostly use innermost (normal order) evaluation; this means that the computation rule is less powerful than lazy evaluation. However, we are more careful about evaluating the arguments of "If," permitting a little more flexibility and some non-strict functions. We obtain programs then as follows:

```
procedure F(x1, ..., xn) % assume F(x1 ... xn) = G(t1 ... tm)
   y1 ← t1(x1 ... xn);
   y2 ← t2(x1 ... xn);
    ...
   ym ← tm(x1 ... xn);
   y ← G(y1, ..., ym);
   return(y)
end F;

procedure A(x);
   return(a)
end A;

procedure If(x,y,z);
   x1 ← x;
   if x1 = true then y1 ← y; return(y1)
     else z1 ← z; return(z1)
end If;

procedure f(x1, ..., xn) % assume f is a constructor
   y1 ← x1 ;
   y2 ← x2 ;
    ...
   yn ← xn ;
   return f(y1, ..., yn)
end f;
 ...
```

Of course, it should be possible to directly generate reasonably efficient programs in machine language, too.

8.8 Lambda calculus

We now briefly give a logic programming language based on the lambda calculus. For this, the logic programs $L(X;Y)$ are conjunctions of equations of the form $Y_i = \alpha_i$ where Y_i is a program variable and α_i is a lambda calculus term, possibly containing variables from X, functions from the underlying logic, and applications of a fixpoint operator $\mu x.t(x)$ which returns a value y such that $t(y) = y$. We prefer a μ operator to the lambda term $(\lambda x.t(xx))(\lambda x.t(xx))$ since the latter seems difficult to type. To show PL-adequacy, for point 2 we have initially $L[X,U;Y,Z]$ of the form $(\wedge_i Y_i = \alpha_i(X,U)) \wedge (\wedge_j(Z_j = \beta_j(X,U)))$. We have $L[X;Y,Y,Z]$ then as $\wedge_i(Y_i = \alpha_i(X,Y)) \wedge (\wedge_j(Z_j = \beta_j(X,Y)))$. We can eliminate the occurrences of Y from the α_i and β_j by a sequence of introductions of the μ operator. For example, from $Y_1 = \alpha_1(X, Y_1, \ldots, Y_n)$ we can obtain that $Y_1 = \mu W.\alpha_1(X, W, Y_2, \ldots, Y_n)$ and then replace all occurrences of Y_1 by $\mu W.\alpha_1(X, W, Y_2, \ldots, Y_n)$. Repeated applications of this idea eliminate all the Y_i, giving us $M[X;Y,Z]$ as required. We need to show that there exists a constructible $h : PC \rightarrow PC$ such that for all F and G in PC, if $(F,G) \models L$ then $(H(F),G) \models M$. This H is obtained by a sequence of fixpoint operations, as indicated above, and these produce another function in PC by properties of programming structures. For point 4 we have the equation $G = t[F]$. For point 5, we have the equation $Y = a$. The inference mechanism is α, β, and η reduction of the lambda calculus, together with the rule $\mu x.t(x) = t(\mu x.t(x))$. These inference rules are sound, by properties of PC.

The interface is equations of the form $(Y_i s_1 \ \ldots \ s_n) = t$, where s_i and t are lambda terms containing the fixpoint operator and possibly constants from the underlying logic. It looks like we could show extensibility by letting μ be a least-fixpoint operator, which guarantees that the specified functions are the least functions satisfying their definitions, and thus are in some sense "initial." Of course, this would require the existence of an appropriate domain structure with least-fixpoints.

We should also assume that functions to compute constructors, destructors, and test for the top constructor symbol are available; this can be done by including some function constants for these functions with the corresponding equations for computing them. Or else this can be done by a suitable encoding into the lambda calculus itself. This lambda calculus approach has some similarity to LISP, and so it might be possible to extend it in that direction. This might give us an approach based on a von Neumann-style architecture, since LISP has efficient implementations on such architectures. Also, the equations $Y_i = \alpha_i$ are remeniscent of assignment statements.

8.9 Derived rules of inference and efficiency

We now go back to the derived rule of inference from section 7.1 and show how it could be realized for term-rewriting systems. We then give a general technique by which such derived rules of inference, with customized logic programs, may be proven correct. Recall that the point is to find more efficient realizations of derived rules of inference, when possible. The rule in question is the following:

$$\frac{[\rightarrow S]P(S,0) \qquad [\rightarrow X](\forall n)(\forall w)(n \geq 0 \wedge P(w,n) \supset P(X(w), n+1))}{[\rightarrow Y](\forall n)(n \geq 0 \supset P(Y(n), n))} \quad (It_1)$$

From the given proof of this rule we obtain a term-rewriting system something like this:

$S \rightarrow \ \ldots$	(rules for computing S)
$X(w) \rightarrow \ \ldots$	(rules for computing X)
$Y(n) \rightarrow (\text{if } n = 0 \text{ then } S \text{ else } X(Y(n-1)))$	(from the proof)

However, we can also use the following system to realize this inference rule:

$S \rightarrow \ \ldots$	(as above)
$X(w) \rightarrow \ \ldots$	(as above)
$Y(n) \rightarrow F(S, n)$	
$F(z, s(v)) \rightarrow F(X(z), v)$	
$F(z, 0) \rightarrow z$	

This uses an iterative rather than a recursive approach, and also a more efficient representation for n; this may be more efficient than the automatically generated program. And of course for more complicated inference rules, the savings obtained by customizing the programs could be much greater, allowing increased efficiency within the framework of a reliable program generation method. Of course, it is also necessary to prove the correctness of the customized program.

We now expand on this, and give a systematic method for proving the correctness of such customized programs.

Theorem 8.19 *A computable mapping $G : L_1 \ \ldots \ L_n \rightarrow L$ from logic programs to logic programs realizes an inference rule*

$$\frac{[X_1 \to Y_1]A_1 \ \ldots \ [X_n \to Y_n]A_n}{[X \to Y]A}$$

if the following inference rule is derivable in PL:

$$\frac{[X_1 \to Y_1']L_1 \ \ldots \ [X_n \to Y_n']L_n}{[X \to Y']G(L_1 \ \ldots \ L_n)}$$

and if the following formula is valid in PL:

$$((\forall X_1 Y_1')(L_1 \supset A_1) \wedge \ \ldots \ \wedge (\forall X_n Y_n')(L_n \supset A_n)) \supset (\forall X Y')(G(L_1 \ \ldots \ L_n) \supset A).$$

Here we assume that Y_i' are the output variables of L_i, and Y' are the output variables of $G(L_1 \ \ldots \ L_n)$, and $Y_i \subset Y_i'$ for all i, and $Y \subset Y'$.

Proof. We need to show that if L_i implements $[X_i \to Y_i]A_i$ for all i, then $G(L_1 \ \ldots \ L_n)$ implements $[X \to Y]A$. For this, we need to show that there is a computable F such that for functions $F_1 \ \ldots \ F_n$ in the program domains satisfying $A_1 \ \ldots \ A_n$, respectively, if F_i satisfies L_i for all i then F satisfies $G(L_1 \ \ldots \ L_n)$. But this is just what is guaranteed by the above inference rule. We also need to know that if $(\forall X_i Y_i')(L_i \supset A_i)$ for all i, then $(\forall X Y')(G(L_1 \ \ldots \ L_n) \supset A)$. This is guaranteed by the above PL formula. Note that the proof of the inference rule given above, will automatically generate another logic program; however, this program is not necessarily the one that is wanted.

□

For our example, we prove the following annotated inference rule:

$$\frac{[L_1(S) :\to S]P(S,0) \quad [L_2(X) :\to X](\forall n)(\forall w)(n \geq 0 \wedge P(w,n) \supset P(X(w), n+1))}{[L_1(S) \wedge L_2(X) \wedge L_3(F,X,Y,S) \to Y](\forall n)(n \geq 0 \supset P(Y(n),n))} \quad (It_1)$$

where $L_3(F,X,Y,S)$ is the following term-rewriting system:

$Y(n) = F(S,n)$
$F(z, s(v)) = F(X(z), v)$
$F(z, 0) = z$

That is, these are equalities which are executed as rewrite rules. Here F, X, Y, and S are all output variables. According to the preceding theorem, in order to prove the correctness of this annotated inference rule we need to derive the following in PL:

$$\frac{[\to S]L_1(S) \quad [\to X]L_2(X)}{[\to F,X,Y,S]L_1(S) \wedge L_2(X) \wedge L_3(F,X,Y,S)}$$

For this, we derive $[F,S \to Y]Y(n) = F(S,n)$ by the functional composition rule with individual functions. We derive $[X \to F]F(z,w) = (\ if\ W = 0\ then\ Z\ else\ F(X(z), w-1))$ by the functional compositional rule with recursion and individual functions. We then need to do a derivation of the following form to put everything together:

$$\frac{[\to S]L_1(S) \quad [\to X]L_2(X) \quad [X \to F]L_3'(FX) \quad [F,S \to Y]L_3''(F,S,Y)}{[\to F,X,Y,S]L_1(S) \wedge L_2(X) \wedge L_3'(F,X) \wedge L_3''(F,S,Y)}$$

This can be done by repeated application of the following inference rule:

$$\frac{[X \to Y]A \quad [Y, W \to Z]B}{[X, W \to Y, Z]A \wedge B} \quad (assuming\ X, Z\ and\ W, Y\ disjoint)$$

This inference rule can be proven as follows:

1.	$[X \to Y]$	$A[X, Y]$	(given)
2.	$[Y, W \to Z]$	$B[Y, W, Z]$	(given) (note that Y, Z and W, Z are disjoint)
3.	$[V, W \to Z]$	$B[V, W, Z]$	(renaming, 2) (so that V, Z are disjoint)
4.	$[X, V, W \to Y, Z]$	$A[X, Y] \wedge B[V, W, Z]$	(combination rule)
			(Y, Z and X, Z and V, Y and W, Y disjoint)
5.	$[X, W \to Y, Z]$	$A[X, Y] \wedge B[Y, W, Z]$	(recursion rule)

We also need to show that the following formula is valid in the underlying logic:

$$((\forall S)(L_1(S) \supset A_1) \wedge (\forall X)(L_2(X) \supset A_2))$$
$$\supset (\forall FXYS)(L_1(S) \wedge L_2(X) \wedge L_3(F, X, Y, S) \supset A)$$

where A_1 is $P(S, 0)$, A_2 is $(\forall n)(\forall w)(n \geq 0 \wedge P(w, n) \supset P(X(w), n+1))$, and A is $(\forall n)(n \geq 0 \supset P(Y(n), n))$. This can be done by an induction on n.

9 Examples of Underlying Logics

We have seen examples of programming languages for PL, and in the process have observed some connections between the evaluation strategy, the derivability relation $\vdash_{LL}$, the semantics, and the underlying logic. The only place where the underlying logic enters the system is the underlying logic rule, so the derivability relation $\vdash_{UL}$ needs to be chosen so that this rule is sound, and this is a function of the semantics. In particular, this depends on the choice of a semantics for PC. Generally, we choose the rest of the logic to be something like the usual language of mathematics (e.g., ZF), so the main variation between underlying logics will typically be in the axioms referring to program construction functions. Now we consider more closely some examples of underlying logics. These are often similar to the programming languages just considered. Note that most of the rules of inference of PL are independent of the underlying logic, as long as it obeys the classical first-order laws with respect to quantifiers. So the only rules that need to be checked for specific underlying logics and program domains and the program construction domains PC are the functional composition rules (where assumed) and the recursion rule.

9.1 Partial recursive functions

Consider for example the underlying logic where the domains are the non-negative integers and for each n, we have a program domain of partial recursive functions with n arguments. The set PC can be specified as the recursive mappings from program domains (partial recursive functions) to program domains, where partial recursive functions are encoded as integers according to some enumeration of the partial recursive functions. We assume that the syntax allows arithmetic operations (at least 0, successor, and predecessor), conditionals, first-order quantifiers, definitions of primitive and partial recursive functions, and maybe more. In this underlying logic we may have a number of axioms, including the assumptions that 0, successor, and predecessor are computable. We also assume that the conditional function defined by $cond(x, y, z) = (if\ x = 0\ then\ y\ else\ z)$ is computable. For this underlying logic we would also include the functional composition rules. These are satisfied because the composition of partial recursive functions is partial recursive.

The recursion rule is satisfied because of the recursion theorem. We note however that the recursion theorem is inconvenient for actual programming; there are much simpler ways of constructing partial recursive functions. For example, we can express partial recursive functions by means of a first-order term-rewriting system, and then one can get the effect of recursion by adding a recursive call as shown in section 8.2.

As an example, we derive other partial recursive functions and show their correctness. For example, we can derive the function $F(x, y)$ defined by (if $(x = y)$ then 1 else 0) as follows:

1. $[\to F]$ $(\forall x, y, z)(x = 0 \supset F(x, y, z) = y) \wedge (x \neq 0 \supset F(x, y, z) = z)$ assumption)
2. $[\to F]$ $(\forall x)(F(x) = x + 1)$ (assumption)
3. $[\to F]$ $(\forall x)(F(x) = x - 1)$ (assumption)
4. $[\to X]$ $(X = 0)$ (assumption)
5. $[X, F, H, K, G' \to G]$ $(\forall xy)(G(x, y) = F(x, F(y, H(X), X), F(y, X, G'(K(x), K(y)))))$ (functional composition)
6. $[G' \to G]$ $(\exists XFHK)(\forall xyz)(X = 0 \wedge F(x, y, z) = (if\ x = 0\ then\ y\ else\ z)$ $\wedge h(x) = x + 1 \wedge K(x) = x - 1$ $\wedge (G(x, y) = F(x, F(y, H(X), X), F(y, X, G'(K(x), K(y))))))$ (renaming rule and several applications of the program composition rule)
7. $[G' \to G]$ $(\forall x, y)G(x, y) = (if\ x = 0\ then\ (if\ y = 0\ then\ 1\ else\ 0)$ $else\ if\ (y = 0\ then\ 0\ else\ G'(x - 1, y - 1)))$ (underlying logic rule)
8. $[\to G]$ $(\forall Xy)G(x, y) = (if\ x = 0\ then\ (if\ y = 0\ then\ 1\ else\ 0)\ else$ $if\ (y = 0\ then\ 0\ else\ G(x - 1, y - 1)))$ (recursion rule)
9. $[\to G]$ $(\forall Xy)\ (x = y \supset G(x, y) = 1 \wedge x \neq y \supset G(x, y) = 0)$ (underlying logic rule)

We now show the programs (expressed as term-rewriting systems over the non-negative integers) for each of the above steps except applications of the underlying logic rule, which do not affect the term-rewriting system:

1. $F_1(0, y, z) \to y$
   $F_1(s(x), y, z) \to z$
2. $F_2(x) \to s(x)$
3. $F_3(s(x)) \to x$
4. $X \to 0$
5. $G(x, y) \to F(x, F(y, H(X), X), F(y, X, G'(K(x), K(y))))$

6. $G(x,y) \rightarrow F_1(x, F_1(y, F_2(X), X), F_1(y, X, G'(F_3(x), F_3(y)))))$
$F_1(0,y,z) \rightarrow y$
$F_1(s(x),y,z) \rightarrow z$
$F_2(x) \rightarrow s(x)$
$F_3(s(x)) \rightarrow x$
$X \rightarrow 0$

8. $G(x,y) \rightarrow F_1(x, F_1(y, F_2(X), X), F_1(y, X, G(F_3(x), F_3(y)))))$
$F_1(0,y,z) \rightarrow y$
$F_1(s(x),y,z) \rightarrow z$
$F_2(x) \rightarrow s(x)$
$F_3(s(x)) \rightarrow x$
$X \rightarrow 0$

The above step shows how the recursion rule affects the program; the call to G' is replaced by a recursive call to G. We note that programs expressed as term-rewriting systems in this way are fairly efficient to execute in many cases. Although this program uses the successor notation, it would be possible to use a more efficient representation for the integers. There are also possibilities for concurrency as in any functional programming language. We can optimize this program by eliminating occurrences of F_2 and X to obtain the following program:

$G(x,y) \rightarrow F_1(x, F_1(y, s(0), 0), F_1(y, 0, G(F_3(x), F_3(y))))$
$F_1(0,y,z) \rightarrow y$
$F_1(s(x),y,z) \rightarrow z$
$F_3(s(x)) \rightarrow x$

This generation of term-rewriting systems has already been formalized in a more general context in section 8.5.

The following theorem is evidence that the recursion mechanism of PL is powerful enough to capture computable recursions.

Theorem 9.1 *Suppose f is a partial recursive function, extended so that $f(x) = \perp$ if f does not terminate on input x. Then the following is deriviable in PL with underlying logic as given above, and with the least fixpoint and first functional composition rule::*

$$[\rightarrow F](\forall x)(F(x) = f(x))$$

Proof. By induction on the length of the definition of f as a partial recursive function. We can imitate recursion, composition, and the minimization operator by appropriate inference rules in PL, especially the least fixpoint rule and the first functional composition rule.

□

9.2 Lambda calculus

Another example would be typed lambda calculus with the μ (least fixpoint) operator. Here the program domains would be continuous mappings from integers (with $\perp$) to integers, and higher sorts based on it. We would need a domain structure for the sorts. The program construction functions PC would be mappings from program domains to a program domain. The syntax of the logic would have typed lambda calculus terms with the μ operator, integers (i.e., zero, successor, and predecessor), the equality predicate, and maybe more. We would interpret the logic in the usual way, with application in the lambda calculus interpreted as functional application on the domains. Also, we interpret $\mu(x)A$ to be the least x (in the domain ordering) such that $Ax = x$. Note that this always exists if the domain has certain necessary properties, since we can take the least upper bound of $A^i(\perp)$. The μ operator guarantees the fixpoint property. In fact, in this case one can use the least fixpoint rule. We can also use the functional composition rules; they are sound because one can compose lambda terms in the lambda calculus.

9.3 Horn clause logic programming

For pure Prolog (sets of Horn clauses) interpreted as in first-order logic, we might have the following "fixpoint" rule:

$$[L \to P](Horn(L) \supset (for\ all\ terms\ x)([(\forall y)L \vdash_{FOL} q(x)] \equiv P(x)))$$

where $Horn(L)$ means L is a set of first-order Horn clauses and $\vdash_{FOL}$ represents derivability in first-order logic and $y = y_1 \ldots y_n$ are the variables in L and q is some predicate in L and $x = x_1 \ldots x_m$ are variables in the query. This says that there is a program to compute logical consequences of sets of Horn clauses. Such a computation may fail (or fail to terminate) on inputs for which P is false; since we haven't formally specified what it means to compute P, this is permissible. The execution mechanism would have to be some complete inference method such as breadth-first search or depth-first iterative deepening; Prolog's depth-first search would not work because it is incomplete. Now, suppose we have a specific logic program L expressing membership in a list:

$$[A \to L]Horn(L) \wedge (for\ all\ terms\ x)[(\forall y)L \vdash_{FOL} member(x)] \equiv x \in list(A)$$

Here $list(A)$ is a list and A is a program that outputs this list. A can be for example a functional program that outputs the desired list, or a program which, given an integer i, gives the i^{th} element of a list. Then, from this rule and the above Prolog fixpoint rule, we can obtain the following by an application of program composition:

$$[A \to P](\exists L)(Horn(L) \supset (for\ all\ terms\ x)([(\forall y)L \vdash_{FOL} member(x)] \equiv P(x))$$

$$\wedge Horn(L) \wedge (for\ all\ terms\ x)([(\forall y)L \vdash_{FOL} member(x)] \equiv x \in list(A)))$$

By an application of the underlying logic rule we obtain

$$[A \to P](P(x) \equiv x \in list(A))$$

Thus by a roundabout route we have constructed a proof, which can be effectively converted into a program that will test for membership in a list $list(A)$. This program would execute by applying some general inference mechanism or complete execution strategy to the set of Horn clauses expressing membership in a list.

We now give a couple of more examples of programs in different formalisms. It would also be interesting to do this for the unification algorithm.

9.4 Derivation of Factorial program

We now show how a proof corresponding to the factorial function can be derived in this system, where the underlying logic is assumed to be the partial recursive functions, with initially only a few functions assumed to be computable. In particular, we start out with the assumptions that a conditional function, the constants 0 and 1, subtracting one, and multiplication are computable. Using the functional composition rule with individual functions we obtain the formula

1. $[X_1 \to Y](\forall x)(Y(x) = if\ x = 0\ then\ 1\ else\ x * X_1(x-1)))$

Now using the underlying logic rule we obtain

2. $[X_1 \to Y](\forall n)((\forall x)(0 \leq x \leq n-1 \supset X_1(x) = fact(x)) \supset (\forall x)(0 \leq x \leq n \supset Y(x) = fact(x)))$

Using the recursion rule we obtain

3. $[\to Y](\forall n)((\forall x)(0 \leq x \leq n-1 \supset Y(x) = fact(x)) \supset (\forall x)(0 \leq x \leq n \supset Y(x) = fact(x)))$

Using the underlying logic rule (mathematical induction) we get

4. $[\to Y](\forall x)(0 \leq x \supset Y(x) = fact(x))$

This is a specification of the factorial function, with individual functions for the zero test, multiplication, decrementing by one, and the conditional test. Assuming the use of term-rewriting systems to express the underlying programs as above, we obtain the following sequence of programs:

1. $Y(x) \to (if\ x = 0\ then\ s(0)\ else\ x * X_1(x-1))$
   ... definition of multiplication ...
   $(if\ 0 = 0\ then\ y\ else\ z) \to y$
   $(if\ s(x) = 0\ then\ y\ else\ z) \to z$
   $s(x) - 1 \to x$

(The definitions of multiplication etc. come from the assumptions that these are computable. These are by-products of the derivation of the functional composition rule with individual functions.)

3. $Y(x) \to (if\ x = 0\ then\ s(0)\ else\ x * Y(x-1))$
   ... definition of multiplication ...
   $(if\ 0 = 0\ then\ y\ else\ z) \to y$
   $(if\ s(x) = 0\ then\ y\ else\ z) \to z$
   $s(x) - 1 \to x$

We note that multiplication could be defined in terms of the successor function, which is rather slow. Or we could substitute a more efficient definition and a more efficient representation of the integers. A sample computation of $Y(2)$ using outermost (lazy) rewriting would be

$Y(s(s(0))) \to if\ s(s(0)) = 0\ then\ s(0)\ else\ s(s(0)) * Y(s(s(0)) - 1) \to s(s(0)) * Y(s(s(0)) - 1) \to s(s(0)) * (if\ s(0) = 0\ then\ s(0)\ else\ s(0) * Y(s(0) - 1)) \to s(s(0)) * (s(0) * Y(s(0) - 1)) \to s(s(0)) * (s(0) * (if\ 0 = 0\ then\ s(0)\ else\ 0 * Y(0-1))) \to s(s(0)) * (s(0) * s(0)) \to \ldots \to s(s(0)) * s(0) \to \ldots \to s(s(0))$.

The proof could have been obtained faster by a use of the functional composition rule with recursion, as follows:

1.	$[\to Y]$	$(\forall x)(Y(x) = if\ x = 0\ then\ 1\ else\ x * Y(x-1))$ (functional composition rule with recursion)
2.	$[\to Y]$	$(\forall x)(0 \leq x \supset Y(x) = fact(x))$ (underlying logic rule)

If we had instead started with a formula

$$[X_1 X_2 \to Y](\forall x)(Y(x) = if\ x = 0\ then\ 1\ else\ X_2(x, X_1(x-1))))$$

we would end up with the formula

$$[X_2 \to Y]((\forall x, y)(X_2(x, y) = x * y) \supset (\forall x)(0 \le x \supset Y(x) = fact(x)))$$

The proof of this formula corresponds to the factorial function with the multiplication parameterized. Thus, an arbitrary proof that $X_2(x, y) = x * y$ can be substituted to obtain a version of the factorial function, corresponding to the use of an arbitrary procedure for multiplying integers. In this way we obtain a kind of abstract data structures. The program (term rewriting system) corresponding to this formula would have no definition for the function X_2 and so could not be directly executed.

9.5 Termination

We now sketch how it is possible to reason about termination in this system. For this, we again use the factorial function. We assume that the integers are extended with a "bottom" ($\perp$) element indicating nontermination and that the functions are defined also on $\perp$ in a reasonable way. We show that $x \ge 0 \supset Y(x) \ne \perp$ in the specification of the factorial function. That is, we assume that this statement is added to the specification of the factorial function and attempt to derive a function satisfying this specification. This is done as before, except that this extra assertion is proved using mathematical induction in an application of the underlying logic rule, since $Y(0) = 1$ and if $Y(x) \ne \perp$ then $Y(x+1) \ne \perp$. We would like to show that if $x < 0$ then $Y(x) = \perp$. For this, we need to use the least fixpoint rule. Then we consider the function $Z(x)$ defined as (if $x < 0$ then $\perp$ else $fact(x)$). We show that this is also a fixpoint of the given specification. Therefore the least fixpoint (least in the domain ordering $<_d$) must be no larger. This implies that if $x < 0$ then $Y(x) = \perp$. It should be clear (in principle) how this can be extended to many examples using arbitrary well-founded orderings and arbitrary least-fixpoints of functions, to prove both termination and nontermination.

9.6 Term-rewriting systems

We give an example of a derivation of a program involving term-rewriting systems. Here we are proving properties of term-rewriting systems at a meta-level, instead of using them as the computational mechanism. First we assume the computability of the one-step derivability relation for a term-rewriting system R. By taking a suitable fixpoint, we obtain a program to compute the result of arbitrarily many rewrites. We then show that if R is terminating and confluent, this program computes normal forms. Next we construct a program to decide if $s =_R t$ for terms s, t where $=_R$ is the underlying equality theory. We use $comp\_reduction(Y)$ as an abbreviation for the formula (if s is R-irreducible then $Y(s) = s$ else $s \to_R Y(s)$).

1. $[Y, Z \to W]$ $(\forall u)(W(u) = Z(Y(u)))$
(functional composition)

2. $[Y, Z \to W]$ $(if\ s\ is\ R\ irreducible\ then\ Y(s) = s\ else\ s \to_R Y(s))$
$implies\ (\forall u)(W(u) = Z(Y(u)))$
(underlying logic rule)

3. $[Y \to W]$ $comp\_reduction(Y)\ implies\ (\forall u)(W(u) = W(Y(u)))$
(recursion rule)

4. $[Y \to W]$ $comp\_reduction(Y) \wedge (R\ terminating\ and\ confluent)$
$implies\ (\forall u)(W(u) =\ normal\ form\ of\ u)$
(underlying logic rule)

5. $[Y \to W]$ $comp\_reduction(Y) \wedge (R\ terminating\ and\ confluent)$
$implies\ (\forall u, v)(W(u) = normalformofu$
$\wedge W(v) = normalformofv)$
(underlying logic rule)

6. $[Y \to W]$ $comp\_reduction(Y) \wedge (R\ terminating\ and\ confluent)$
$implies\ (\forall u, v)(W(u) = W(v)) \equiv (u =_R v)$
(underlying logic rule, Church Rosser property)

7. $[W \to W']$ $(\forall u, v)(W'(u, v) = (if\ W(u) = W(v)\ then\ true\ else\ false))$
(computability of conditional, assumed)

8. $[Y \to W']$ $(\exists W) comp\_reduction(Y) \wedge (R\ terminating\ and\ confluent)$
$implies\ (\forall u, v)(W(u) = W(v)) \equiv (u =_R v)$
$\wedge (\forall u, v)(W'(u, v) = (if W(u) = W(v)\ then\ true\ else\ false))$
(composition rule)

9. $[Y \to W']$ $comp\_reduction(Y) \wedge (R\ terminating\ and\ confluent)\ implies$
$(\forall u, v)(W'(u, v) \equiv (u =_R v))$
(underlying logic rule)

Thus we have derived a function W' to decide the equational theory of R, given that R is terminating and confluent and that some one-step rewrite relation is computable.

10 Goal-Directed Derivations

Many program generation systems permit a program to be derived in a systematic way from its specification. We make some general comments about how this can also be done with the current approach. We present a collection of derived inference rules for facilitating goal-directed program generation. These rules may also be useful in a more general context, that is, for arbitrary applications of the logic PL. These rules are not necessarily original with us, just formalized in a different framework. The idea is that we are given a formula $A(X, Y)$ expressing the desired relationship between the inputs X and the outputs Y of the desired program. By backward reasoning, we typically derive a collection of formulae $B_i(t_i(X), Z)$ and conditions $C_i(X)$ such that $C_i(X) \wedge B_i(t_i(X), Z) \supset A(X, r_i(X, Z))$. Also, the C_i, t_i,and r_i are assumed to be computable. The idea is that we can test C_i, and if it is true, recursively compute Z such that $B_i(t_i(X), Z)$ and then return $r_i(X, Z)$ as the output. The B_i are then additional specifications that must be computed in a similar way, and hopefully we can obtain enough recursions so that the computation can be carried out. Also, we would like to have that $(\forall X)(C_1(X) \vee \ldots \vee C_n(X))$. For our formalism, we introduce a function variable F intended to represent a function computing a value (or sequence of values) $F(X)$ such that $(\forall X)A(X, F(X))$. We then derive the formula $[\to F](\forall X)A(X, F(X))$ in our system, which will construct an F as desired.

10.1 General subgoaling principle

We first give a general scheme for goal-directed program generation based on a specification and properties of a function or program at the top level. We can express this as follows.

Suppose we are trying to prove $[\rightarrow F](\forall x)P(F,x)$. We can start with the valid formula $[G \rightarrow F](\forall x)(P(G,x) \supset P(F,x))$. Then we can begin a process of subgoaling, working on the leftmost $P(G,x)$ and eventually eliminating it altogether. In general, this process of subgoaling operates as follows: In a formula of the form $[Y_1 \ldots Y_n \rightarrow F](\forall x)(B_1(Y_1,x) \wedge \ldots \wedge B_n(Y_n,x) \supset P(F,x))$, we regard the $B_i(Y_i,x)$ as subgoals and attempt to replace them by something simpler and eventually eliminate them. For this we can use the following rule:

$$\frac{\begin{array}{c}[X_1 \rightarrow Y_1](\forall x)A_1(X_1,x) \supset B_1(Y_1,x) \ \ldots \ [X_n \rightarrow Y_n](\forall x)A_n(X_n,x) \supset B_n(Y_n,x) \\ [Y_1 \ \ldots \ Y_n \rightarrow F](\forall x)(B_1(Y_1,x) \wedge \ \ldots \ \wedge B_n(Y_n,x) \supset P(F,x))\end{array}}{[X_1 \ \ldots \ X_n \rightarrow F](\forall x)(A_1(X_1,x) \wedge \ \ldots \ \wedge A_n(X_n,x) \supset P(F,x))} \quad (S_1)$$

Here X_i and Y_i can be single program variables or sequences of program variables. This rule operates on all of the subgoals B_i at once, and allows us to replace B_i by A_i. A special case is when A_i is "true", and then B_i is eliminated. Another special case is when A_i and B_i are identical; then B_i is unchanged. We use this rule in a forward direction, but it has the effect of backward reasoning. This can be proven using the composition rule a number of times.

We now give another version of this principle that is related to the underlying logic rule.

$$\frac{\begin{array}{c}(\forall x, X_1)A_1(X_1,x) \supset B_1(X_1,x) \ \ldots \ (\forall x, X_n)A_n(X_n,x) \supset B_n(X_n,x) \\ [X_1 \ \ldots \ X_n \rightarrow F](\forall x)(B_1(X_1,x) \wedge \ \ldots \ \wedge B_n(X_n,x) \supset P(F,x))\end{array}}{[X_1 \ \ldots \ X_n \rightarrow F](\forall x)(A_1(X_1,x) \wedge \ \ldots \ \wedge A_n(X_n,x) \supset P(F,x))} \quad (S_2)$$

This permits the subgoals B_i to be modified according to the rules of the underlying logic, without considering how X_i is computed. If A_i is true, then B_i has been proved in the underlying logic and can be omitted.

10.2 Booleans and computability

Conditional functions are of the form (if P then A else B) where P is a computable predicate. In order to develop rules for conditionals, therefore, we have to consider more closely the relationship between Booleans and computability. To say that a predicate C is computable means that we can compute a function F such that $C(x) = f(x)$ for all X. We abbreviate this by $comp(C)$. We also use $comp(f)$ where f is a function to abbreviate $[\rightarrow F](F = f)$, and say then that f is computable. In general, if t is a term containing variables $x_1 \ \ldots \ x_n$ then $comp(t)$ abbreviates $[\rightarrow F](\forall x_1 \ \ldots \ x_n)f(x_1 \ \ldots \ x_n) = t$. However, this introduces some difficulties in the system. The problem has to do with $\perp$ and other such elements that one must add to obtain domains with fixpoints, as required by our fixpoint rule. Any computable function or predicate has to be monotone. That implies that if $P(\perp)$ is true then $P(x)$ is true for all x, and if $P(\perp)$ is false then $P(x)$ is false for all x. Therefore, the only non-trivial predicates are those for which $P(\perp)$ is some element other than true and false, typically $\perp$. This means that all of our logical operations must be somehow extended to consider $\perp$, and we have to give up the identity $(\forall x)(P(x) \vee \neg(P(x)))$ or else change the meaning of the connectives in some way to cope with $\perp$. There is also a problem with conditionals, since (if $\perp$ then A else B) is $\perp$.

Our solution to this is to separate the logical formalism from the computational formalism. Since computability is already explicitly accounted for by the $[X \rightarrow Y]$

quantifier, the underlying logic need not be concerned with it. We require that logical connectives and quantifiers only take Boolean values. Program variables, including predicates, may evaluate to non-Booleans like $\perp$. We convert from non-Boolean values to Boolean values by means of the functions $true(A)$, $false(A)$, and $def(A)$, defined respectively by $A = true$, $A = false$, and $(A = true \vee A = false)$. We note that these functions are not computable, since $true(\perp) = false$ but $true(true) = true$, so the function $true$ is not monotone. When a quantifier or logical connective is applied to a non-Boolean, we assume that the function "true" is implicitly applied to the arguments. Note that we have for all A, $true(A) \vee neg(true(A))$ but we do not have $true(A) \vee false(A)$ always. Also note that $A \vee neg(A)$ holds even if A is computable since this implicitly coerces to $true(A) \vee neg(true(A))$. However, $comp(A \wedge B)$ is false if $A \wedge B$ is nontrivial, since A and B are here implicitly converted to Booleans, and thus $A \wedge B$ can nowhere be $\perp$.

10.3 Conditional rules

We now specialize the general rules given above to conditionals, since they are common and have special logical properties. We start with a conditional rule corresponding to a program with a conditional at the top. We assume the function $if(x, y, z)$ is computable and satisfies $(x = true \supset if(x, y, z) = y \wedge x = false \supset if(x, y, z) = z)$.

One plausible (but wrong) conditional rule is

$$\frac{comp(C) \quad comp(if)}{[G_1 G_2 \rightarrow G](\forall x)(C(x) \supset G(x) = G_1(x)) \wedge (\neg C(x) \supset G(x) = G_2(x))}$$

This is "proved" by noting that the conditional $(if\ C(x)\ then\ G_1(x)\ else\ G_2(x))$ satisfies the conclusion and is computable, using the functional composition rule. The problem occurs if G_1 and G_2 are both constant functions (say, 0 and 1) and x is $\perp$. Then $(if\ C(x)\ then\ G_1(x)\ else\ G_2(x))$ evaluates to $\perp$, so $G(x) = G_1(x)$ and $G(x) = G_2(x)$ are both false. We fix this rule as follows:

$$\frac{comp(C) \quad comp(if)}{[G_1 G_2 \rightarrow G](\forall x)(C(x) \supset G(x) = G_1(x)) \wedge (false(C(x)) \supset G(x) = G_2(x))} \quad (C_1)$$

The proof of this corresponds to the construction of the conditional (if $C(x)$ then $G_1(x)$ else $G_2(x)$).

Another conditional rule is as follows:

$$\frac{comp(C) \quad comp(if) \quad [\rightarrow G](\forall x)(C(x) \supset A(G, x)) \quad [\rightarrow G](\forall x)(false(C(x)) \supset A(G, x))}{[\rightarrow F](\forall x)(def(C(x)) \supset A(F, x))} \quad (C_2)$$

However, this rule is only sound for $A(G, x)$ of the form $A'(G(x), x)$. To show that this rule is unsound for general A, let $C(x)$ be $even(x)$ ("x is even") and let $A(G, x)$ be $(even(x) \supset G(x)G(x-1) < 0 \wedge odd(x) \supset G(x)G(x+1) > 0)$. Now, we can satisfy $[\rightarrow G](\forall x)(C(x) \supset A(G, x))$ by taking $G(x)$ as $(if\ \ even(x)\ then\ 1\ else\ -1)$ and we can satisfy $[\rightarrow G](\forall x)(false(C(x)) \supset A(G, x))$ by taking $G(x)$ as 1. However, there is no function F satisfying $[\rightarrow F](\forall x)def(C(x)) \supset A(F, x)$, since such a function would have to satisfy $G(x)G(x-1) < 0$ and $G(x)G(x-1) > 0$ simultaneously for even x. When A depends only on the value of $G(x)$, that is, A is of the form $A'(G(x), x)$ (a common case), then C_2 can be shown by the functional composition rule, noting that if g_1 satisfies $(\forall x)(true(C(x)) \supset A(g_1, x))$ and g_2 satisfies $(\forall x)(false(C(x)) \supset A(g_2, x))$ then $(if\ C(x)\ then\ g_1(x)\ else\ g_2(x))$ satisfies $(\forall x)(def(C(x)) \supset A(F, x))$. A slightly stronger form of this rule, suitable for use with the subgoal rule S_1 given above, is as follows, again assuming that $A(G, x)$ is of the form $A'(G(x), x)$:

$$\frac{comp(C) \quad comp(if)}{[G_1G_2 \to F](\forall x)((C(x) \supset A(G_1,x)) \wedge (false(C(x)) \supset A(G_2,x)) \supset (def(C(x)) \supset A(F,x)))} \quad (C_3)$$

Here the subgoals are $(C(x) \supset A(G_1,x))$ and $(false(C(x)) \supset A(G_2,x))$ and the goal is $def(C(x)) \supset A(F,x)$. This form of the rule is better when x needs to be remembered for purposes of mathematical induction. From now on we assume that the conditions $A(G,x)$ are all of the form $A'(G(x),x)$ and similarly for $B(G,x)$ and $P(G,x)$ (when used).

10.3.1 A multiple choice conditional rule

From now on we assume $comp(if)$ as a hypothesis for all rules, so this condition is often omitted. We extend the above rule to more than one condition as follows:

$$\frac{(\wedge_i comp(C_i)) \quad (\forall x)(C(x) \supset \vee_i C_i(x))}{[G_1 \ \ldots \ G_n \to F](\forall x)(\wedge_i(C(x) \wedge C_i(x) \supset A(G_i,x)) \supset (C(x) \supset A(F,x)))}$$

Assuming the hypotheses, this rule creates subgoals $(C(x) \wedge C_i(x) \supset A(G_i,x))$ of the goal $C(x) \supset A(F,x)$; these can be used with the above subgoaling rule. The problem with this rule is that we need to compute which one of the C_i is true, for a given X. This can only be done by computing them in parallel and waiting until one of them returns "true." however, in general, this kind of parallel computation is non-monotonic. We could retain this rule anyway, since we have computability in an intuitive sense, but the non-monotonicity could cause the fixpoint property to be violated for the constructed function F. Therefore we add a hypothesis, and obtain the following rule:

$$\frac{(\wedge_i comp(C_i)) \quad (\forall x)(C(x) \supset \vee_i C_i(x)) \ (\wedge_i(C(x) \supset def(C_i(x))))}{[G_1 \ \ldots \ G_n \to F](\forall x)(\wedge_i(C(x) \wedge C_i(x) \supset A(G_i,x)) \supset (C(x) \supset A(F,x)))} \quad (CM_1)$$

Of course this rule also has the following alternative form:

$$\frac{\begin{array}{c}(\wedge_i comp(C_i)) \quad (\forall x)(C(x) \supset \vee_i C_i(X)) \\ (\wedge_i(C(x) \supset def(C_i(x)))) \quad \wedge_i [\to G_i](\forall x)(C(x) \wedge C_i(x) \supset A(G_i,x))\end{array}}{[\to F](\forall x)(C(x) \supset A(F,x))} \quad (CM_2)$$

Using the second deduction principle and some logical rewriting, we obtain another form that will be used later:

$$\frac{\begin{array}{c}(\wedge_i comp(C_i)) \quad (\forall x)(C(x) \supset \vee_i C_i(X)) \\ (\wedge_i(\forall x)(C(x) \supset def(C_i(x)))) \quad \wedge_i [F \to G_i](\forall x)(C(x) \wedge C_i(x) \supset (B(F) \supset A(G_i,x)))\end{array}}{[F \to G](\forall x)(C(x) \supset (B(F) \supset A(G,x)))} \quad (CM_3)$$

10.3.2 A rule for term structure and conditionals

The conclusion of the multiple choice conditional rule CM_1 has subgoals $(C(x) \wedge C_i(x) \supset A(G_i,x))$; used with the subgoaling principle S_1, these correspond to assumptions of the form $[X_i \to G_i](\forall x)(A_i(X_i,x) \supset (C(x) \wedge C_i(x) \supset A(G_i,x)))$ that permit the subgoal $(C(x) \wedge C_i(x) \supset A(G_i,x))$ to be replaced by $A_i(X_i,x)$. We give a rule that can generate such formulas:

$$\frac{(\forall f)(\forall x)(A_i(f,t_i(x)) \supset (C_i(x) \supset A(u_i(f,x),x))) \quad comp(u_i)}{[X \to G](\forall x)(A_i(X,t_i(x)) \supset (C_i(x) \supset A(G(x),x)))} \quad (CT_1)$$

For example, since we know that $(\forall f,x)(f(x-1) = fact(x-1) \supset (x > 0 \supset x * fact(x-1) = fact(x)))$, we can derive $[X \to G](\forall x)(X(x-1) = fact(x-1) \supset (x > 0 \supset G(x) = fact(x)))$.

10.4 Functional composition and subgoaling

We now give a rule that permits the elimination of computable functions at the top of an expression. This may be used together with the subgoaling rule S_1. Recall that comp(f) is an abbreviation for $[\to F](F = f)$.

$$[Z_1 \ldots Z_n Z \to Y](\forall x)[(C(x) \supset Z_1(x) = t_1) \wedge \ldots \wedge (C(x) \supset Z_n(x) = t_n) \wedge (C(x) \supset Z(t_1 \ldots t_n) = f(t_1 \ldots t_n)) \supset (C(x) \supset Y(x) = f(t_1 \ldots t_n))] \quad (F_1)$$

If F is computable then the rule is a little simpler:

$$\frac{comp(f)}{\begin{array}{c}[Z_1 \ldots Z_n \to Y](\forall x)[(C(x) \supset Z_1(x) = t_1) \wedge \ldots \wedge (C(x) \supset Z_n(x) = t_n) \\ \supset (C(x) \supset Y(x) = f(t_1 \ldots t_n))]\end{array}} \quad (F_2)$$

This rule is also suitable for use with the subgoal principle S_1. The goal $(C(x) \supset Y(x) = f(t_1 \ldots t_n))$ generates the subgoal $(C(x) \supset Z(t_1 \ldots t_n) = f(t_1 \ldots t_n))$ and the subgoals $(C(x) \supset Z_i(x) = t_i)$. Here x can be a sequence of variables. We can push $(\forall x)$ in and split up the hypotheses if desired. We prove this rule by the underlying logic (for equality) and functional composition rules. Using the composition rule, we can derive a related rule as follows:

$$\frac{comp(f) \quad [\to Z_1](\forall x)(C(x) \supset Z_1(x) = t_1) \ldots [\to Z_n](\forall x)(C(x) \supset Z_n(x) = t_n)}{[\to Y](\forall x)(C(x) \supset Y(x) = f(t_1 \ldots t_n))} \quad (F_3)$$

10.5 Mathematical induction

We now derive the following inference rule:

$$\frac{\begin{array}{c}comp(x > 0) \quad [\to G_1]A(G_1(0), 0) \\ [F \to G_2](\forall x)(x > 0 \wedge A(F(x-1), x-1) \supset A(G_2(x), x))\end{array}}{[\to G](\forall x)(x \geq 0 \supset A(G(x), x))} \quad (I_1)$$

1. $[\to G_1]$ $A(G_1(0), 0)$
   (assumed)
2. $[F \to G_2]$ $(\forall x)(x > 0 \wedge A(F(x-1), x-1) \supset A(G_2(x), x))$
   (assumed)
3. $[G_1 G_2 \to G]$ $(\forall x)(x > 0 \supset G(x) = G_2(x)) \wedge (false(x > 0) \supset G(x) = G_1(x))$
   (using conditional rule C_1, since $x > 0$ is computable)
4. $[F \to G]$ $(\exists G_1 G_2)(\forall x)(A(G_1(0), 0) \wedge (x > 0 \wedge A(F(x-1), x-1) \supset A(G_2(x), x))$
   $\wedge (x > 0 \supset G(x) = G_2(x)) \wedge (false(x > 0) \supset G(x) = G_1(x)))$
   (the program composition rule, 1,2,3)
5. $[F \to G]$ $(\forall x)((x = 0) \supset A(G(x), 0) \wedge ((x > 0) \supset A(F(x-1), x-1) \supset A(G(x), x)))$
   (underlying logic rule)
6. $[F \to G]$ $(\forall x)(A(G(0), 0) \wedge ((x \geq 0) \supset A(F(x), x) \supset A(G(x+1), x+1)))$
   (rewriting a little)
7. $[\to G]$ $(\forall x)(A(G(0), 0) \wedge ((x \geq 0) \supset A(G(x), x) \supset A(G(x+1), x+1)))$
   (recursion rule)
8. $[\to G]$ $(\forall x)(x \geq 0 \supset A(G(x), x))$
   (induction and the underlying logic rule)

Note that if x is $\perp$ then $x > 0$ is undefined, but the induction hypotheses is $x \geq 0 \supset A(G(x), x)$, that is, $true(x \geq 0) \supset A(G(x), x)$, so only defined elements are considered. The term $x - 1$ in this rule can be replaced by any other term $t(x)$ satisfying $(x > 0 \supset t(x) < x)$:

$$\frac{\begin{array}{c} comp(x>0) \quad (x>0 \supset t(x)<x) \quad [\to G_1]A(G_1(0),0) \\ [F \to G_2](\forall x)(x>0 \wedge A(F(t(x)),t(x)) \supset A(G_2(x),x)) \end{array}}{[\to G](\forall x)(x \geq 0 \supset A(G(x),x))} \quad (I_2)$$

Assuming as usual that $A(F,x)$ is of the form $A'(F(x),x)$, we can write this as follows:

$$\frac{\begin{array}{c} comp(x>0) \quad (x>0 \supset t(x)<x) \quad [\to G_1]A(G_1,0) \\ [F \to G_2](\forall x)(x>0 \wedge A(F,t(x)) \supset A(G_2,x)) \end{array}}{[\to G](\forall x)(x \geq 0 \supset A(G,x))} \quad (I_3)$$

We can generalize this to an arbitrary well-founded ordering as follows:

$$\frac{[F \to G](\forall x)((\forall y)(y<x \supset A(F,y)) \supset A(G,x)) \qquad < \ well-founded}{[\to G](\forall x)A(G,x)} \quad (I_4)$$

Here we can let x be a tuple of variables as well as a single variable. By $<$ well founded, we mean that the ordering $true(x<y)$ is well-founded. Note that if "$x<\perp$" is undefined always, then the hypotheses imply $A(G,\perp)$. Thus A must be stated in a way so as to include such undefined elements.

10.6 Induction and conditionals

Combining the above rule I_4 with the conditional rule CM_3, we obtain the following rule that combines conditionals and induction:

$$\frac{\begin{array}{c} (\wedge_i comp(C_i)) \quad (\forall x)(C(x) \supset \vee_i C_i(x)) \quad (\forall x,i)(C(x) \supset def(C_i(x))) \\ \wedge_i [F \to G_i](\forall x)(C(x) \wedge C_i(x) \supset (\forall y)(y<x \supset A(F,y)) \supset A(G,x)) \qquad < \ well-founded \end{array}}{[\to G](\forall x)(C(x) \supset A(G,x))} \quad (IC_1)$$

This is proven as follows: By the conditional rule CM_3 given above, we conclude from the hypotheses that $[F \to G](\forall x)(C(x) \supset (\forall y)(y<x \supset A(F,y)) \supset A(G,x))$. By the above well-founded induction rule I_4, we obtain $[\to G](\forall x)(C(x) \supset A(G,x))$. We now have the problem of how to prove the hypotheses $[F \to G_i](\forall x)(C(x) \wedge C_i(x) \supset (\forall y)(y<x \supset A(F,y)) \supset A(G,x))$. For this we can use the following rule:

$$\frac{\begin{array}{c} (\forall x,F)(C_i(x) \supset (\wedge_i A(F(t_i(x)),t_i(x)) \supset A(t(F(t_1(x)), \ \ldots, \ F(t_n(x))),x))) \\ (\forall x)(C_i(x) \supset t_i(x)<x) \quad comp(t_i) \quad comp(t) \end{array}}{[F \to G](\forall x)(C_i(x) \supset (\forall y)(y<x \supset A(F(y),y)) \supset A(G(x),x))} \quad (IC_2)$$

The proof is by taking $G(x)$ to be $t(F(t_1(x)), \ \ldots, \ F(t_n(x)))$; in our system, we can prove this using a functional composition rule.

10.6.1 Constructor rules

The rule I_1 can also be generalized to constructors; for this we have the following rule:

$$\frac{comp(x>0) \quad [\to G_1]A(G_1,0) \quad [F \to G_2](\forall x)(x \geq 0 \wedge A(F,x) \supset A(G_2,s(x)))}{[\to G](\forall x)(x \geq 0 \supset A(G,x))} \quad (Co_1)$$

This is more appropriate because the successor function $s(x)$ suggests a constructor. We now consider the case of lists in more detail. For this we have the constructor "cons" and the destructors "car" and "cdr." It is often the case that functions on lists are obtained by computing the first element of the list and then doing some recursion to obtain the tail of the list; thus, $F(cons(x,y)) = cons(t(x,y), F(u(x,y)))$ where t, u are computable. We can express this in an inference rule as follows, where $<$ orders lists by length:

$$\frac{\begin{array}{c} comp(empty(x)) \quad comp(u) \quad comp(t) \quad [\rightarrow G_1](\forall x)(empty(x) \supset A(G_1, x)) \\ (\forall F, x)(false(empty(x)) \wedge A(F, u(x)) \supset A(cons(t(x), F(u(x))), x)) \\ (\forall x)(false(empty(x)) \supset u(x) < x \wedge t(x) < x) \end{array}}{[\rightarrow G](\forall x)(list(x) \supset A(G, x))} \quad (L_1)$$

Note that empty($\perp$) will be $\perp$, and empty(x) in the hypotheses is coerced to true(empty(x)). Here list(x) is a predicate that is true of all lists without $\perp$. The proof of this rule involves induction rules developed above (especially IC_1). It should be clear that many more such rules could be derived as needed.

We now develop a constructor rule for arbitrarily many constructors. Let $T(\mathcal{F}, \mathcal{X})$ be the set of terms over a set $\mathcal{F}$ of function symbols and a set $\mathcal{X}$ of variables. We have the following rule:

$$\frac{\begin{array}{c} comp(s \in \mathcal{X}) \quad [\rightarrow F](\forall s \in \mathcal{X})A(F, s) \\ [F \rightarrow G](\forall f(s_1 \ \ldots \ s_n) \in T(\mathcal{F}, \mathcal{X}))((\wedge_i A(F, s_i)) \supset A(F, f(s_1 \ \ldots \ s_m))) \end{array}}{[\rightarrow F](\forall s, t \in T(\mathcal{F}, \mathcal{X}))A(F, s)} \quad (Co_2)$$

This may be proven using rule I_4 or IC_1, since the depth ordering on finite terms is well-founded. We now consider the case of many constructors and two arguments; this is useful for the unification function synthesis. We have the following rule:

$$\frac{\begin{array}{c} comp(s \in \mathcal{X}) \quad [\rightarrow F](\forall s \in \mathcal{X})(\forall t \in T(\mathcal{F}, \mathcal{X}))A(F, s, t) \\ [\rightarrow F](\forall t \in \mathcal{X})(\forall s \in T(\mathcal{F}, \mathcal{X}))A(F, s, t) \\ [F \rightarrow G](\forall f(s_1 \ \ldots \ s_m), g(t_1 \ \ldots \ t_n) \in T(\mathcal{F}, \mathcal{X})) \\ ((\wedge i, j A(F, s_i, t_j)) \supset A(F, f(s_1 \ \ldots \ s_m), g(t_1 \ \ldots \ t_n))) \end{array}}{[\rightarrow F](\forall s, t \in T(\mathcal{F}, \mathcal{X}))A(F, s, t)} \quad (Co_3)$$

Here we are assuming $A(F, s, t)$ is of the form $A'(F(s), F(t))$ for some A', as usual. This is also proven using I_4 or IC_1, where pairs $< s, t >$ of terms are ordered by the sum of their depths; this ordering is well-founded for pairs of finite terms.

Of course, one could also develop rules based on I_4 or IC_1 for more than one term, that perform induction on the sum of their depths; this would be a generalization of the above rule Co_3 and is often useful too.

10.6.2 Induction and goal-directed program construction

We comment on how these rules can be used in goal-directed program generation. When attempting to derive $[\rightarrow G](\forall x)(x \geq 0 \supset A(G, x))$, the cases $x = 0$ and $x > 0$ are likely to be generated automatically. The case $x = 0$ may be solved first, and then the subgoal remains to solve the case $x > 0$. This may automatically generate a subgoal of the form $A(F, t(x)))$; in our approach (using rules S_1 and S_2) this is represented by the formula $[F \rightarrow G_2](\forall x)(x > 0 \wedge A(F, t(x)) \supset A(G_2, x))$. When such a formula is generated, it suggests to see if $t(x) < x$ and if the induction principle can be applied. For this it is only necessary to generate the formula $[\rightarrow G_1]A(G_1, 0)$ and one application of the derived rule for induction generates the conclusion $[\rightarrow G](\forall x)(x \geq 0 \supset A(G, x))$. Thus this fits into our general proof strategy and also gives a method to automatically detect when induction should be attempted. Or, if we solve the case $x = 0$ first, this may suggest to try an induction, although this gives less guidance about how it should be done.

In fact we can adapt the subgoal principle S_1 to an arbitrary rule of inference; this corresponds to a proof transformation of the following type:

If we have a proof of the form

$$\frac{[X_1 \to Z_1](\forall x)A_1(X_1, Z_1) \ \ldots \ [X_m \to Z_m](\forall x)A_m(X_m, Z_m)}{[\to Y](\forall x)B_1(Y_1, x)}$$

(possibly an induction proof) then we can derive a rule of the form

$$\frac{\begin{array}{c}[X_1 \to Z_1](\forall x)A_1(X_1, Z_1) \ \ldots \ [X_m \to Z_m](\forall x)A_m(X_m, Z_m) \\ [Y_1 \ \ldots \ Y_n \to F](\forall x)(B_1(Y_1, x) \wedge \ \ldots \ \wedge B_n(Y_n, x) \supset P(F, x))\end{array}}{[Y_2 \ \ldots \ Y_n \to F](\forall x)(B_2(Y_2, x) \wedge \ \ldots \ \wedge B_n(Y_n, x) \supset P(F, x))}$$

Although formally fairly trivial, this shows us for example how the induction rules can be used together with the goal-directed approach.

10.6.3 A well-founded recursion rule

We now give another combination of an induction principle with a conditional rule; this one essentially eliminates all computable functions from an expression at once.

$$\frac{\begin{array}{c}[F \to G](\forall x)C_i(x) \wedge A(F, t_i(x)) \supset A(G, x) \quad [F \to G](\forall x)D_i(x) \supset A(G, x) \\ comp(C_i) \quad comp(D_i) \\ (\forall x)(C(x) \supset (\vee_i C_i(x)) \vee (\vee_i D_i(x))) \\ (\forall x)t_i(x) < x \quad < \ well\ founded \\ (\forall x, i)(C(x) \supset def(C_i(x))) \quad (\forall x, i)(C(x) \supset def(D_i(x)))\end{array}}{[\to G](\forall x)(C(x) \supset A(G, x))} \quad (R_1)$$

This rule may seem somewhat unwieldy. But it can be used in a fairly natural way. The idea is to find (computable) conditions C_i and D_i, where the C_i permit G to be computed recursively and the D_i permit G to be computed nonrecursively. These conditions should be exhaustive and should permit induction on some well founded ordering. We can think of t_i as the argument of recursive calls of F; these are important for induction. For example, for factorial we have $D_1(x)$ as $x = 0$ and $C_1(x)$ as $x > 0$. We have $t_1(x)$ as $x - 1$. We note that the recursive call of $fact(x)$ is on $x - 1$, which is smaller in the usual ordering on the natural numbers. We can also do mutual recursion by having more than one program variable F and G. We can prove this rule R_1 using the inference rules IC_1 and IC_2. From IC_2 we can derive $[F \to G](\forall x)(C_i(x) \supset (\forall y)(y < x \supset A(F(y), y)) \supset A(G(x), x))$ and $[F \to G](\forall x)(D_i(x) \supset (\forall y)(y < x \supset A(F(y), y)) \supset A(G(x), x))$. Then from IC_1 we can derive $[\to G](\forall x)A(G, x)$.

10.7 Examples

We now give some examples of goal directed program generation. For these examples we assume that the underlying logic is the usual language of mathematics, possibly Zermelo-Fraenkel set theory. First we give a goal-directed derivation of the factorial function. We start with the goal $[\to F](\forall x)(x \geq 0 \supset F(x) = fact(x))$. The condition we use initially is $x = 0$; using the rule C_2, this generates the subgoals $[\to F](\forall x)(x = 0 \supset F(x) = fact(x))$ and $[\to F](\forall x)(x > 0 \supset F(x) = fact(x))$. The first may be rewritten using the underlying logic rule to $[\to F](\forall x)(x = 0 \supset F(x) = fact(0))$ which simplifies to $[\to F](\forall x)(x = 0 \supset F(x) = 1)$; this can generate the subgoal $[\to F](\forall x)F(x) = 1$. Since 1 is computable, this subgoal is solved. We now return to the subgoal $[\to F](\forall x)(x > 0 \supset F(x) = fact(x))$. Using the underlying logic rule, this generates the subgoal $[\to F](\forall x)(x > 0 \supset F(x) = x * fact(x-1))$. Since $*$ is computable, we generate from rule F_2 the two subgoals $[\to F](\forall x)(x > 0 \supset F(x) = x)$ and $[\to F](\forall x)(x > 0 \supset F(x) = fact(x-1))$. The first is solvable since the identity is computable. The second generates the subgoals $[\to F](\forall x)(x > 0 \supset F(x-1) = fact(x-1))$ and $[\to F](\forall x)(x > 0 \supset F(x) = x-1)$. The second is easily solved since $-$, 1, and x are computable. The first remains unsolved. It is significant incidentally that the argument to F in the first subgoal is $x-1$ and not x; otherwise, the induction would require more intelligence to discover. The rule F_1 was carefully constructed to generate this argument automatically.

Gathering all this together, we have shown that $[G \to F](\forall x)((x > 0 \supset G(x-1) = fact(x-1)) \supset (x \geq 0 \supset F(x) = fact(x)))$. (This could have been shown formally using the above rules, but it is simpler to give an informal proof.) This fits into the format of our induction rule I_1 and so it only remains to prove the subgoal $[\to F](\forall x)(x = 0 \supset F(x) = fact(x))$ (which has already been proven). We note that it was necessary to remember the value of x so that this induction could be done. The rules (especially the rules S_1 and S_2) were structured to make this possible.

The formal structure of this proof is as follows:

1. $[G \to F]$ $(\forall x)[(x \geq 0 \supset F(x) = fact(x)) \supset (x \geq 0 \supset F(x) = fact(x))]$
   (underlying logic rule, computability of identity)
2. $[G \to F]$ $(\forall x)[(x > 0 \supset G(x-1) = fact(x-1)) \supset (x \geq 0 \supset F(x) = fact(x))]$
   (1, by proof as sketched above, doing subgoaling)
3. $[\to F]$ $(\forall x)(x = 0 \supset F(x) = fact(x))$
   (as shown above)
4. $[\to F]$ $(\forall x)(x \geq 0 \supset F(x) = fact(x))$
   (2,3, induction rule I_1)

We now derive a simple sorting algorithm in a goal-directed manner. We chose an algorithm complex enough so that some nontrivial reasoning is involved in the derivation. We represent lists as usual, and use car, cdr, and cons as in LISP. Suppose the specification $A(F, x)$ is $sorted(F(x)) \wedge mset(F(x)) = mset(x)$, where $mset(x)$ is the multiset of elements of a list and $sorted(x)$ means x is empty or $car(x) = min(x)$ and $sorted(cdr(x))$. Also, $min(x)$ is the smallest element in a list x. We start with the condition (x is empty) generating the subgoals (x is empty and $A(F, x)$) and (x is not empty and $A(F, x)$). The first is easily solved since all empty lists are sorted. This suggests to use an induction principle. From the induction principle IC_1, it suffices to show $[G \to F](\forall x)(\neg empty(x) \supset (A(G, u(x)) \supset A(F, x)))$ for some $u(x)$ smaller than x. However, we choose to use the even more specific rule L_1 for lists given earlier:

$$\frac{\begin{array}{c} comp(empty(x)) \quad comp(u) \quad comp(t) \quad [\to F_1](\forall x)(empty(x) \supset A(F_1, x)) \\ (\forall G, x)(\neg empty(x) \wedge A(G, u(x)) \supset A(cons(t(x), G(u(x))), x)) \\ (\forall x)(\neg empty(x) \supset (u(x) < x) \wedge t(x) < x) \end{array}}{[\to F](\forall x)(list(x) \supset A(F, x))}$$

Here $<$ orders lists by their length. From the proof of this we know that $\neg empty(x) \supset F(x) = cons(t(x), G(u(x)))$. We need to show $(\forall G, x)(\neg empty(x) \wedge A(G, u(x)) \supset A(F(x), x))$. Recall that $A(F, x)$ is $sorted(F(x)) \wedge mset(F(x)) = mset(x)$ for non-empty x. Also, $sorted(y)$ is $car(y) = min(y) \wedge sorted(cdr(y))$ then. Therefore, replacing $sorted(F(x))$ by its definition, $A(F, x)$ is $car(F(x)) = min(F(x)) \wedge sorted(cdr(F(x))) \wedge mset(F(x)) = mset(x)$. Since $car(F(x)) = min(F(x))$ and $mset(F(x)) = mset(x)$, $car(F(x)) = min(x)$. Since $F(x) = cons(t(x), G(u(x)))$, $t(x) = min(x)$ and $G(u(x)) = cdr(F(x))$. To determine $u(x)$, we try to derive properties of $cdr(F(x))$. We can derive that $mset(F(x))$ is the union of $\{car(F(x))\}$ and $mset(cdr(F(x)))$; thus $mset(cdr(F(x))) = mset(F(x)) - mset(car(F(x))) = mset(x) - mset(min(x))$. The expression $mset(x) - \{min(x)\}$ involves removing an element from a multiset. We want to use computable functions on lists instead of functions like "$-$" on multisets. Let $remove(x,e)$ be a function that removes one of the occurrences of e from x, if x occurs in e. As a subgoal (not shown here) we can derive that such a function is computable. (We also can derive that $min(x)$ is computable.) Thus we obtain $mset(cdr(F(x))) = mset(remove(x, min(x)))$. Now, $A(G, u(x))$ is $sorted(cdr(F(x))) \wedge mset(cdr(F(x))) = mset(u(x))$ since $G(u(x)) = cdr(F(x))$. This suggests that $u(x)$ is $remove(x, min(x))$, and with this we can derive $(\forall G, x)(\neg empty(x) \wedge A(G, u(x)) \supset A(cons(t(x), G(u(x))), x))$ as required. Noting that $u(x) < x$ and $t(x) < x$ and u and t are computable, the proof is completed.

Another easy example of a program that can be derived this way is $sublist(x, y)$ that checks that every element in x appears in y; this can be expressed (for non-empty x) as $element(car(x), y) \wedge sublist(cdr(x), y)$. This fits a modification of rule I_4 (or IC_1) in which the well-founded ordering is the sum of the term depths. It can also be regarded as a modified induction principle L_1 for lists (regarding x as the argument of the function) with "cons" replaced by an arbitrary computable function.

We now discuss the most general unifier function, since it is a frequent test case for program generation methods. Our method permits much of the derivation to be done in the underlying logic; thus the proof becomes relatively short, but possibly not representative of the difficulty of deriving the program. Also, part of the proof is already done in the derivation of the derived inference rules (especially Co_3). We sketch the derivation here. Let $[\rightarrow F](\forall s, t \in T(\mathcal{F}, \mathcal{X}))mgu(s,t) = F(s,t)$ be our specification, where $mgu(s,t)$ is a most-general unifier of terms s and t in $T(\mathcal{F}, \mathcal{X})$ if they are unifiable, else "fail." Using the constructor rule Co_3, we obtain the subgoals $[\rightarrow F](\forall s \in \mathcal{X})mgu(s,t) = F(s,t)$, $[\rightarrow F](\forall t \in \mathcal{X})mgu(s,t) = F(s,t)$, and $mgu(f(s_1 \ldots s_m), g(t_1 \ldots t_n)) = F(f(s_1 \ldots s_m), g(t_1, \ldots, t_n))$. The first two subgoals are solved by noting that if $s = t$, then $mgu(s,t) = \{\}$, else if s occurs in t, then $mgu(s,t) =$ "fail", else $mgu(s,t) = \{s \leftarrow t\}$. (This can be derived in the underlying logic.) Thus we can derive $[\rightarrow F]mgu(s,t) = F(s,t)$ using a conditional rule, assuming the occurrence check is computable. Of course, that must be shown as a lemma, which is easy using the rule Co_3 again. For the last subgoal, we use the fact that the most general unifier of $f(s_1 \ldots s_m)$ and $g(t_1 \ldots t_n)$ is fail, if $f \neq g$, and otherwise it does not depend on f or g. This also can all be shown in the underlying logic. So we have a function $mgu\_list([s_1, \ldots, s_m], [t_1, \ldots, t_n])$ to compute the most general (simultaneous) unifier of these two lists and the fact that $mgu(f(s_1 \ldots s_n), f(t_1 \ldots t_n)) = mgu\_list([s_1, \ldots, s_m], [t_1, \ldots, t_n])$. We can then apply Co_3 again, or use our above scheme for list computations (extended to two arguments) to find a simple recursive computation of $mgu\_list([s_1 \ldots s_m], [t_1 \ldots t_n])$ in terms of $mgu\_list([s_2 \ldots s_m], [t_2 \ldots t_n])$. This requires methods for composing and applying substitutions, which also have to be derived in PL.

11 Deriving Programs versus Deriving Outputs

We note that the fixpoint property requires that the program variables represent programs and not specific inputs and outputs. Thus in this formalism it is not convenient to derive statements like $(\forall x)(\exists y)A(x,y)$ where x are input variables and y are output variables. Such statements can be derived in the above formalism if y is computable by a finite straight-line program, but it is difficult or impossible (we think) to do if the computation of A requires recursion. This is because the recursion rule is difficult to apply in a natural way at this level. For example, it's not typical recursion to look for an x such that $x = x^2 + 5$. We note that it is theoretically possible to find a (real) x such that $x = x^2 + 5$; we can let x be "bottom." Or we can let x be a complex number. However, this is not usually what is wanted when reasoning about recursion and inputs and outputs in this manner. Instead, we state the specification in the form $[\rightarrow Z](\forall x)A(x, Z(x))$. In this way, it becomes more natural to do recursion since it is more natural to look for a program Z such that $Z = F(Z)$ where F is some program-forming operation. We have demonstrated above that recursive programs can easily be generated in this way. In fact, the fact that we can generate all partial recursive functions is a kind of evidence of completeness of this approach. Also, it is easy to show that if a program can be verified within a certain formalism (such as Floyd's method) then we can construct a proof representing that program. This probably also applies to Hoare-style logics.

12 Higher-Order Features

It is possible to get much of the power of this system in a first-order logic. The program variables may be higher order variables, but it is not necessary. They may be individual (first-order) variables. This allows the underlying logic to be first-order. This is reasonable since the progam variables refer to programs, which are textual objects (character strings). We can then indicate the application of a program variable X to arguments $t_1 \ldots t_n$ by $f_{eval}(X, t_1 \ldots t_n)$ or $P_{eval}(X, t_1 \ldots t_n)$ where f_{eval} is a function and P_{eval} is a predicate, depending on the sort of X. Also, if we are reasoning about several kinds of semantics, then we can use several such f_{eval} and P_{eval} functions together.

We note that the program variables typically represent first-order functions or predicates. These can be informally regarded as of sort $[input^m \rightarrow output]$. The program construction functions then represent higher-order functions, since they map tuples of programs to programs. These are therefore informally of sort PC, which can approximately be written as $[program^n \rightarrow program]$. A proof can be regarded as a way to obtain new program construction functions, so it can be regarded as a yet higher-order function which permits the program-construction functions in the hypothesis to be mapped onto the program-construction functions in the conclusion. A proof would then have sort something like $[PC^p \rightarrow PC]$. By considering deduction rules of the form "If A is derivable from B then C is derivable from D" we can obtain still higher-order such functions. Therefore, even though the logic does not explicitly specify methods for obtaining arbitrarily high order programs, it does give some implicit mechanism for doing this.

However, there is a more direct way to incorporate higher order features into this logic. For this we have to extend the definition of a programming structure so that PC also contains computable functions from PC to PC. The idea is to permit the quantifier $[X \rightarrow Y]$ to be more general, for example, something like $[(X \rightarrow Y) \rightarrow (U \rightarrow V)]$, signifying a function from $X \rightarrow Y$ to $U \rightarrow V$. The semantic formula corresponding to $[(X \rightarrow Y) \rightarrow (U \rightarrow V)]A(X, Y, U, V)$ would then be $(\exists G)(\forall XFU)A(X, F(X), U, G(F)(U))$. Thus G maps functions $X \rightarrow Y$ to functions $U \rightarrow V$. Such quantifiers (analogous to types) could be arbitrarily complicated, as, $[(X \rightarrow Y) \rightarrow (U \rightarrow (V \rightarrow W))]$. This feature may be compatible with certain forms of polymorphism, too. As an example of reasoning with this kind of logic, we may have a rule that says that if we can derive $[U \rightarrow V]B$ from $[X \rightarrow Y]A$ then we can conclude $[(X \rightarrow Y) \rightarrow (U \rightarrow V)](A \supset B)$. Also, we might allow a program variable X in $[X \rightarrow Y]A(X, Y)$ to be instantiated by $U \rightarrow V$ to obtain $[(U \rightarrow V) \rightarrow Y]A(F, Y)$ where F maps the sort of U to the sort of V. We have not needed this facility so far, and so we have not presented it in much detail. Also, we can have inference rules as follows:

$$\frac{[X \rightarrow Y]A(X, Y)}{[\rightarrow F](\forall X)A(X, F(X))}$$

$$\frac{[\rightarrow F](\forall X)A(X, F(X))}{[X \rightarrow Y]A(X, Y)}$$

The first rule permits us to go from the level of reasoning about inputs and outputs, to the level of reasoning about programs (and higher levels). The second rule permits us to go from the level of reasoning about programs to the level of reasoning about inputs and outputs. These would have to be restricted so that the sorts of the variables X,Y, and F are all program sorts. Using these rules, we can for example reason about inputs and outputs, and then go up to the level of reasoning about programs in order to do a recursion, which may not be possible at the lower level.

References

[1] Penny Anderson. *Program Derivation by Proof Transformation.* PhD thesis, Carnegie-Mellon University, October 1993.

[2] Philippe Audebaud. Partial objects in the calculus of constructions. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 86–95, Amsterdam, The Netherlands, July 15–18 1991.

[3] David Basin. Isawhelk: Whelk interpreted in Isabelle. Abstract accepted at the 11th International Conference on Logic Programming (ICLP94). Full version available via anonymous ftp to mpi-sb.mpg.de in pub/papers/conferences/Basin-ICLP94.dvi.Z.

[4] David Basin, Alan Bundy, Ina Kraan, and Sean Matthews. A framework for program development based on schematic proof. In *7th International Workshop on Software Specification and Design*, Los Angeles, December 1993. IEEE Computer Society Press.

[5] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[6] W. Bibel. Syntax-directed, semantics-supported program synthesis. *Artificial Intelligence*, 14:243–261, 1980.

[7] R.J. Brachman and J.G. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[8] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[9] Rod M. Burstall and Joseph A. Goguen. The semantics of clear, a specification language. In *Advanced Course on Abstract Software Specification.* LNCS 86, 1980.

[10] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11:197–261, 1988/89.

[11] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[12] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, pages 95–120, 1988.

[13] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North-Holland, 1990.

[14] Gilles Dowek, Amy Felty, Hugo Herbelin, Gerard Huet, Christine Paulin-Mohring, and Benjamin Werner. The coq proof assistant user's guide. Technical report, INRIA, Rocquencourt, France, December 1991.

[15] J.Y. Girard. A new constructive logic: classical logic. *Mathematical Structures in Computer Science*, 1:255–296, 1991.

[16] C.A. Goad. *Computational Uses of the Manipulation of Formal Proofs*. PhD thesis, Stanford University, 1980.

[17] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[18] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the IJCAI-69*, pages 219–239, 1969.

[19] T. Griffin. A formula-as-types notion of control. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 17-19 1990.

[20] W. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 479–490. Academic Press, NY, 1980.

[21] Ina Kraan, David Basin, and Alan Bundy. Logic program synthesis via proof planning. In K.K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, pages 1–14. Springer-Verlag, 1993. Also available as Max-Planck-Institut für Informatik Research Paper MPI-I-92-244.

[22] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. 2nd edn.

[23] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

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

[25] C.R. Murthy. An evaluation semantics for classical proofs. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 96–107, Amsterdam, The Netherlands, July 15–18 1991.

[26] M. Parigot. Free deduction: an analysis of computations in classical logic. In *Proc. Russian Conference on Logic Programming*, pages 361–380, St. Petersburg, Russia, 1991.

[27] M. Parigot. Lambda-mu calculus: an algorithmic interpretation of classical natural deduction. In *Proc. International Conference on Logic Programming and Automated Reasoning*, pages 190–201, St. Petersburg, Russia, 1992.

[28] M. Parigot. Strong normalization for second order classical natural deduction. In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 39–46, Montreal, Canada, June 19-23 1993.

[29] Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, Cambridge University Computer Laboratory, Cambridge, January 1993.

[30] D. Plaisted. Equational reasoning and term rewriting systems. In D. Gabbay and J. Siekmann, editors, *Handbook of Logic in AI and Logic Programming*, volume 1. Oxford University Press, 1993.

[31] Scott F. Smith. *Partial Objects in Type Theory*. PhD thesis, Cornell, 1988.

INFORMATIK

Below you find a list of the most recent technical reports of the research group *Logic of Programming* at the Max-Planck-Institut für Informatik. They are available by anonymous ftp from our ftp server `ftp.mpi-sb.mpg.de` under the directory pub/papers/reports. If you have any questions concerning ftp access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Regina Kraemer
Im Stadtwald
D-66123 Saarbrücken
GERMANY
e-mail: `kraemer@mpi-sb.mpg.de`

MPI-I-94-230	H. J. Ohlbach	Temporal Logic Proceedings of the ICTL Workshop
MPI-I-94-228	H. J. Ohlbach	Computer Support for the Development and Investigation of Logics
MPI-I-94-226	H. J. Ohlbach, D. Gabbay, D. Plaisted	Killer Transformations
MPI-I-94-225	H. J. Ohlbach	Synthesizing Semantics for Extensions of Propositional Logic
MPI-I-94-224	H. Aït-Kaci, M. Hanus, J. J. M. Navarro	Integration of Declarative Paradigms Proceedings of the ICLP'94 Post-Conference Workshop Santa Margherita Ligure, Italy
MPI-I-94-223	D. M. Gabbay	LDS – Labelled Deductive Systems Volume 1 — Foundations
MPI-I-94-218	D. A. Basin	Logic Frameworks for Logic Programs
MPI-I-94-216	P. Barth	Linear 0-1 Inequalities and Extended Clauses
MPI-I-94-209	D. A. Basin, T. Walsh	Termination Orderings for Rippling
MPI-I-94-208	M. Jäger	A probabilistic extension of terminological logics
MPI-I-94-207	A. Bockmayr	Cutting planes in constraint logic programming
MPI-I-94-201	M. Hanus	The Integration of Functions into Logic Programming: A Survey
MPI-I-93-267	L. Bachmair, H. Ganzinger	Associative–Commutative Superposition
MPI-I-93-265	W. Charatonik, L. Pacholski	Negativ set constraints: an easy proof of decidability
MPI-I-93-264	Y. Dimopoulos, A. Torres	Graph theoretical structures in logic programs and default theories
MPI-I-93-260	D. Cvetković	The logic of preference and decision supporting systems
MPI-I-93-257	J. Stuber	Computing Stable Models by Program Transformation
MPI-I-93-256	P. Johann, R. Socher	Solving simplifications ordering constraints
MPI-I-93-250	L. Bachmair, H. Ganzinger	Ordered Chaining for Total Orderings
MPI-I-93-249	L. Bachmair, H. Ganzinger	Rewrite Techniques for Transitive Relations