Denotational and Operational Semantics for Prolog†

Saumya K. Debray
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Prateek Mishra
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794

Abstract: The semantics of Prolog programs is usually given in terms of the model theory of first order logic. However, this does not adequately characterize the computational behavior of Prolog programs. Prolog implementations typically use a sequential evaluation strategy based on the textual order of clauses and literals in a program, as well as non-logical features like ‘‘cut’’. In this work we develop a denotational semantics that captures the computational behavior of Prolog. We present a semantics for ‘‘cut-free’’ Prolog, which is then extended to Prolog with cut. For each case we develop a congruence proof that relates the semantics to a standard operational interpreter. As an application of our denotational semantics, we show the correctness of some standard ‘‘folk’’ theorems regarding transformations on Prolog programs.

---

† A preliminary version of this paper appears in the Proceedings of the IFIP Conference on Formal Description of Programming Concepts, Ebberup, Denmark, Aug. 1986.

This work was supported in part by the National Science Foundation under grant number DCR-8407688.

1. Introduction

Any attempt at formulating a semantics for the programming language Prolog must cope with a certain schizophrenia. From one perspective the question is simply closed, as programs in Prolog are statements in the Horn clause fragment of first-order logic. The semantics of Prolog can therefore be stated in terms of the model theory of first order logic [1, 13]. This is usually referred to as the *declarative* or *logical* semantics of Prolog. From a computational point of view, this formulation is inadequate as it ignores several behavioral aspects of Prolog programs. These include issues such as termination, the use of a sequential depth-first search strategy, and constructs for controlling search, such as *cut*. Further, in practice, Prolog seems to be used more as a language for defining computations over sequences of substitutions than as a language for asserting the truth of certain formulae (e.g. see [10]).

In this work we develop a denotational semantics [12] for Prolog that can express behavioral properties of interest. The semantics incorporates the sequential evaluation strategy used by standard Prolog evaluators and can express the effect of the *cut* operator. A natural consequence is that the meaning of a predicate is a function from substitutions to a (potentially) infinite sequence of substitutions, rather than a set of ground atoms. The reasonableness of the denotational semantics is demonstrated by developing a congruence proof that relates it to an operational interpreter for Prolog.

Our work is motivated by the need to verify various optimizing transformations on Prolog programs. The literature contains various references to folk theorems as the basis for such transformations [4, 11]. Many such theorems entail reasoning about the computational behavior of Prolog programs: e.g., reasoning about termination owing to the insertion or removal of cuts or about the behavior of predicates when viewed as substitution sequence transformers. In Section 5 we use our semantics to give simple proofs for two nontrivial theorems regarding program transformations involving cuts. We are currently using our denotational semantics to validate sophisticated static analysis schemes such as mode inference and determinacy inference. While these theorems could have been proved by reasoning about the behavior of an interpreter, using computational induction or fixpoint induction, this would be difficult because the interpreter does not support reasoning about sequences of substitutions: instead, it is organized around the concept of encoding such information in its state. We believe the denotational approach we use is substantially simpler and more understandable.

Proofs of semantic equivalence are traditionally unreadable, and need be developed with care to possess even a modest degree of versimilitude. The key issue in our proof is reconciling a compositional denotational semantics with an operational semantics that is much more oriented towards ‘‘substitution-at-a-time’’ processing. For the cut-free case we reconcile the two semantic specifications by developing a set of theorems that support a modular decomposition of interpreter states. Unfortunately, for full Prolog such a straightforward decomposition is not possible because of the non-local nature of *cut*, and the proof is instead expressed as an invariant relating the interpreter state to the denotational semantics (Section 4). For the sake of continuity, the proofs of theorems have been relegated to the appendices.

Related work on the denotational treatment of the semantics of logic programming languages includes that of Frandsen [5, 6] and Jones and Mycroft [7]. Frandsen treats ‘‘pure’’ programs and ignores the sequential nature of Prolog's computation, which makes it difficult to use his semantics to explain behavioral aspects of programs. There are a number of differences between our work and that of Jones and Mycroft. Our semantic definitions are motivated by the need to justify program analysis and transformation methods, whereas their definitions are driven by the goal of generating correct Prolog interpreters. In contrast to their direct semantics wherein ''cut'' is a modelled by means of a special token, we give a continuation semantics that models ''cut'' in a more intuitively accessible manner. In final contrast, we provide a congruence proof relating operational and denotational semantics, and use the semantics to validate two nontrivial optimizing transformations of Prolog programs.

The remainder of this paper is organized as follows: Section 2 discusses some basic notions, and develops the notation used in the rest of the paper. Section 3 is concerned with the semantics of cut-free Prolog. Section 4 discusses the semantics of Prolog with cut. Section 5 applies the semantics to validate two optimizing transformations of Prolog programs, and Section 6 concludes the paper. Appendices 1 and 2 contain the proofs of equivalence between the denotational and operational semantics for the cut-free and the cut cases respectively.

2. Preliminaries

2.1. SLD-resolution

A term in Prolog is either a variable, a constant or a compound term $f(t_1, ..., t_n)$ where f is an n-ary function symbol and the t_i, $1 \le i \le n$, are terms. The set of variables, function symbols and predicate symbols will be denoted by **Var**, **Func** and **Pred** respectively. The set of terms will be denoted **Term**. A substitution is an idempotent mapping from **Var** to **Term** which is the identity mapping at all but finitely many points. The set of substitutions will be denoted by **Subst**. Given substitutions σ_1 and σ_2, σ_1 will be said to be *more general* than σ_2 if there is a substitution θ such that $\sigma_2 = \theta \circ \sigma_1$. Two terms t_1 and t_2 are said to be *unifiable* if there exists a substitution σ such that $\sigma(t_1) = \sigma(t_2)$; in this case, the substitution σ is said to be a *unifier* for the terms. If two terms t_1 and t_2 have a unifier, then they have a *most general unifier* $\mathrm{mgu}(t_1, t_2)$, which is unique up to variable renaming.

A Prolog program consists of a set of predicate definitions. A predicate definition consists of a sequence of clauses. Each clause is a sequence of literals, which are either atomic goals or negations of atomic goals. Prolog clauses are generally constrained to be definite Horn, i.e. have exactly one positive literal; the positive literal is called the *head* of the clause, and the remaining literals, if any, constitute the *body* of the clause. A clause with only negative literals is referred to as a *negative clause* or *goal*. We will adhere to the syntax of Edinburgh Prolog and write clauses in the form

$$\mathrm{p} :\!- \mathrm{q}_1, \ldots, \mathrm{q}_n.$$

which can be read as ''p *if* q_1 *and . . . and* q_n''.

The evaluation strategy used by Prolog is an instance of a more general theorem proving procedure called SLD-resolution [1]. An SLD-derivation with respect to a set of clauses P is a sequence N_0, N_1, ...

of negative clauses such that for each i, if $N_i = a_1, ..., a_{n_i}$, then

$$N_{i+1} = \theta(a_1, ..., a_{k-1}, (b_1, ..., b_m), a_{k+1}, ..., a_{n_i})$$

satisfying:

(1) $1 \le k \le n_i$;

(2) b :– $b_1, ..., b_m$ is a clause in P, appropriately renamed to have no variables in common with those in N_i;

(3) θ is the most general unifier of a_k and b.

A program *$langle P,G$rangle* is a set of predicate definitions P togther with a goal clause G. Given a program *$langle P, G$rangle*, an SLD-tree is a tree whose nodes are labelled with goal clauses, with the root being labelled by G. If a node N in the SLD-tree has label $a_1, ..., a_m$, then an atom a_k, $1 \le k \le m$, is selected, and for each clause a' :– b' of P whose head unifies with a_k after appropriate renaming of variables, there is a son of N labelled with the resulting goal.

2.2. Sequential Prolog

An SLD-tree represents a collection of SLD-derivations all starting with q. An execution of a program can be thought of as a traversal of an SLD-tree for that program searching for refutations, i.e. paths that terminate in the empty clause. It can be shown that that the existence of a node labelled by the empty clause in any SLD-tree for a program implies the existence of a node labelled by the empty clause in every SLD-tree for that program [1]. Hence in solving for a refutation it suffices to search any one SLD-tree. Prolog's execution strategy corresponds to a depth-first search of an SLD-tree where the selected atom in a goal is always the leftmost one, and where the sons of a node in the tree are ordered according to the textual order of the corresponding clauses in the program. In such a scheme, the invocation of an atomic goal is handled by unifying it with the appropriate clauses chosen in sequence. This can result in programs that are declaratively identical but computationally distinct. As an example, consider the following definitions of the reverse function:

```
append([],X,X).
append([A|B],Y,[A|D]) :– append(B,Y,D).

rev1([],[]).
rev1([A|X],Y) :– rev1(X,Z), append(Z,[A],Y).

rev2([],[]).
rev2([A|X],Y) :– append(Z,[A],Y), rev2(X,Z).
```

The reader can verify that the goal *rev1*([1,2],X) terminates with X bound to [2,1], whereas the goal *rev2*([1,2],X) yields one solution and then diverges.

2.3. On the Observable Behaviour of Prolog Programs

An aspect of Prolog that has attracted much attention is its ability to define computations over substitutions. A naive semantic definition of the substitution generated by evaluating a goal is to compose the substitutions generated in going from one node in the SLD-tree to the next until an empty clause is encountered. This neglects the fact that the only *observable* aspect of a Prolog computation is in terms of the variables contained in the top-level goal. Our semantic definitions incorporate this notion of observability and discard unnecessary information from substitutions. As an example, the predicates *foo1* and *foo2* have identical meaning in our formulation:

```
foo1(X,Y) :– foo(X,Z).
foo([],[]).

foo2([],Y).
```

2.4. On the Semantics of Cut

The action of encountering a cut during a sequential traversal of a SLD-tree can be explained as follows. By parent goal we will mean the goal that activated the clause under consideration. When a cut is encountered in the clause, the sub-tree of the SLD-tree below the parent goal that has not been traversed by the interpreter is discarded. In effect, this means that (*i*) all but the first substitution obtained from the atoms to the left of the cut are discarded and (*ii*) any substitutions that might have been obtained from the clauses following the current clause in textual order are discarded. In modelling the action of a cut our semantics incorporates this viewpoint explicitly.

2.5. Definitions and Notation

Given a (countable) set S, we use $S_{\$bottom}$ to denote the set $S \cup \{\$bottom\}$. The set of finite and (ω-)infinite sequences of elements of a set S will be denoted by S^{∞}. The set of finite sequences of elements of a set S will be denoted by S^*. The sequence whose first element is a, and where the sequence formed by the second element onwards is L, will sometimes be written $a :: L$; the empty sequence will be denoted by *nil*. Given two sequences S_1 and S_2, their concatenation will be denoted by $S_1\ \$sometimes\ S_2$. Let $\$bottom$ denote the undefined sequence. Then, $\$sometimes$ is defined as follows:

$\$bottom\ \$sometimes\ L = \$bottom$;
$nil\ \$sometimes\ L = L$;
$(a :: L_1)\ \$sometimes\ L_2 = a :: (L_1\ \$sometimes\ L_2)$.

An n-tuple of objects $a_1, ..., a_n$ will be denoted by $\$langle\, a_1, ..., a_n \$rangle$. Where there is no need to distinguish between the elements of a tuple, we will sometimes write the tuple with a bar, e.g. $\bar{t}$.

A common operation on sequences is to apply a function to each element of the sequence, and collect the results in order. This is denoted by ‘||’:

$f || \$bottom = \$bottom$;

$f \| nil = nil$;

$f \| (a :: L) = f(a) :: f \| L$.

Finally, we define an operator to ‘‘collect’’ the results of applying a sequence-valued function to the elements of a sequence of values. Let $f : S_{\$bottom} \rightarrow S^{\infty}_{\$bottom}$ be such a function, and $s \in S^{\infty}_{\$bottom}$ a sequence of S-objects. We wish to apply f to each element of s in order, and concatenate the resulting sequences together to produce the output. This operation is denoted by $\$circle$:

$f\ \$circle\ \ \$bottom = \$bottom$;

$f\ \$circle\ \ nil = nil$;

$f\ \$circle\ \ (a :: L) = f(a)\ \$sometimes\ \ (f\ \$circle\ \ L)$.

The syntactic categories of Prolog are as follows:

Variable ::= **Var**;

Functor ::= **Func**;

Predicate ::= **Pred**;

Term ::= **Term**;

Term ::= *Variable* + *Functor*($Term_1$, ..., $Term_n$);

Atom ::= *Predicate*($Term_1$, ..., $Term_n$);

Literal ::= *Atom* + ‘!’;

Goal ::= $Literal^*$;

Clause ::= *Atom* :– *Goal*.

Recall that a substitution is an idempotent, almost-identity mapping from **Var** to **Term**. Given a substitution σ, we will usually wish to know the effect of σ on some finite set of variables only, which are of interest to us. For this, we define the set **FSubst** to be the set of *finite substitutions*, which are mappings from **Var** to **Term** with finite domain and range. It is convenient to require that any variable ‘‘mentioned’’ in a finite substitution be in its domain. Let the function $vars : \mathbf{Term} \rightarrow 2^{\mathbf{Var}}$ yield the set of variables occurring in a term (extending in the natural way to literals and clauses). Then, we have the following definition:

Definition: A *finite substitution* ϕ is an idempotent mapping from a finite set of variables V to a finite set of terms T, such that for any v in V, $vars(\phi(v)) \subseteq V$.

The domain of a finite substitution ϕ will be written $dom(\phi)$. Given a finite substitution ϕ and a set of variables V containing $dom(\phi)$, the *extension* of ϕ to V, written $\phi \uparrow V$, is the mapping with domain V which is the same as ϕ on $dom(\phi)$, and the identity mapping on $V - dom(\phi)$. Given two finite substitutions θ and σ, their composition $\theta \circ \sigma$ is a finite substitution with domain $dom(\theta) \cup dom(\sigma)$. In the limit, the extension of a finite substitution to the set of all variables **Var** yields a substitution. We will sometimes not distinguish between finite substitutions and substitutions when the context is unambiguous. The set of finite substitutions is closed under finite composition, but not under infinite composition.

The function *unify* : **Term** $\times$ **Term** $\rightarrow$ (**FSubst** $\cup$ {*fail*}) returns the most general unifier of two terms, if it exists, and the token *fail* otherwise. It extends in the natural way to atoms. The function *rename* : **Term** $\times 2^{\mathbf{Var}} \rightarrow$ **Term** takes a term t and a set of variables V and yields a term t' identical to t but with its variables consistently replaced by ‘‘new’’ variables not occurring in V. (We assume that the set of variables **Var** is a countably infinite set v_1, v_2, ... indexed by the natural numbers. Then, given any finite set of variables V, it is possible to effectively find the element $v_k \in V$ which has the largest index of any element in V, and thence to find a variable v_m with $m > k$, so that v_m *$nomem* V. We do not elaborate the nature of *rename* further.) When referring to *rename*(t, V), we will say that t has been ‘‘renamed with respect to V’’. The function *rename* extends in the natural way to literals and clauses.

In the development that follows, we will be interested in the set $\mathbf{FSubst}^{\infty}_{\$bottom}$, the set of finite and infinite sequences of finite substitutions together with the undefined sequence. Elements of **FSubst** will be written as lower case Greek letters σ, θ, ϕ, ..., while elements of $\mathbf{FSubst}^{\infty}_{\$bottom}$ will be written as upper case Greek letters Σ, Θ, Φ, ... The finite substitution with domain V which is the identity mapping will be written ε_V.

Let V be the set of variables occurring in a goal G. If θ is an ‘‘answer substitution’’ for G, then we are interested only in the part of θ dealing with variables in V. Another way of saying this is the observation that since θ is an answer substitution, any instance of $\theta(G)$ is refutable from the program, which implies that for any substitution θ' that agrees with θ on elements of V, $\theta'(G)$ is also refutable from the program. The ‘‘interesting’’ part of θ is therefore the part dealing with V. We use the projection function $\downarrow$: **FSubst** $\times 2^{\mathbf{Var}} \rightarrow$ **FSubst** to restrict a finite substitution to a set of variables:

Definition: Given a finite substitution $\phi : V \rightarrow T$, and $W \subseteq V$, the *projection* of ϕ on W, written $\phi \downarrow W$, is the finite substitution with domain $V_1 = W \cup (\bigcup_{v \in W} vars(\phi(v)))$, such that for any v in V_1, $(\phi \downarrow W)(v)$ = *if* $v \in W$ *then* $\phi(v)$ *else* v.

3. Semantics of Prolog without Cut

3.1. Denotational Semantics for Cut-free Prolog

The meaning of a predicate definition is a function that maps terms to sequences of finite substitutions. The set $\mathbf{FSubst}^{\infty}_{\$bottom}$, of finite and infinite sequences of finite substitutions augmented with a least object *$bottom*, is a complete partial order under the standard prefix ordering. Predicate symbols are associated with their meanings in the usual way, using environments. Intuitively, an environment would map identifier names to functions from terms to substitution sequences. It turns out, however, that for technical reasons (renaming of variables before resolution) it is also necessary to dynamically propagate the set of variables encountered during execution. This is done by passing in, as an argument, the appropriate finite substitution, and performing renaming with respect to the domain of this substitution.

We therefore have

$$Env = Predicate \rightarrow Term \rightarrow \mathbf{FSubst} \rightarrow \mathbf{FSubst}^{\infty}_{\$bottom}.$$

Environments will typically be denoted as ρ, ρ_0, ρ_1, ... The act of binding an identifier *Id* to an object *Obj* in an environment will be indicated by $Id \leftarrow Obj$. Given an environment ρ, the environment $\rho[p \leftarrow f]$ will denote the environment which is the same as ρ except for the value of p, which is f.

The process of defining the denotational semantics of a program can be seen as consisting of two parts. First, the meanings of individual predicates have to be defined in terms of the meanings of their components; this is done using the semantic function **D**[[ ]], which defines the meaning of a sequence of clauses to be a function over environments. Given a sequence of clauses C^* and an environment ρ, $\mathbf{D}[[C^*]]\rho$ gives the new environment obtained by elaborating the clause sequence C^* in environment ρ. Second, the meanings of goals with respect to these definitions have to be specified. This is done using the semantic function **G**[[ ]]: given a sequence of literals G, an environment ρ and an input substitution σ, $\mathbf{G}[[G]]\rho\,\sigma$ gives the output substitution sequence obtained by evaluating $\sigma(G)$ in environment ρ. In fact, the two parts are not independent, since the first part involves defining the meaning of a clause, and defining the meaning of the body of a clause requires the semantic function **G**[[ ]]. We use two auxiliary functions, **C**[[ ]], which gives the meaning of a single clause, and **L**[[ ]], which gives the meaning of a single literal in the body of a clause.

The function **D**[[ ]] takes a clause sequence and an environment and returns a new environment. This can be written in curried form as

$$\mathbf{D}[[\ ]] : Clause^* \rightarrow Env \rightarrow Env.$$

Intuitively, the idea behind **D**[[ ]] is that given some knowledge of what other predicates mean (as specified by the input environment), we can extend our knowledge of those meanings by evaluating the given clause sequence in that environment; this results in a new environment, which is ‘‘better defined’’ than the input environment. An empty sequence of clauses defines nothing, so we have

$$\mathbf{D}[[nil]]\rho = \lambda x \lambda y \lambda z.nil.$$

where x represents an input predicate name, y a tuple of terms and z a substitution.

If, on the other hand, we have a sequence of clauses $c_0 :: C$, then according to Prolog's computation strategy, all the answers that can be generated using clause c_0 are generated before any of the clauses C are attempted. Thus, if the substitution sequence obtained using only cl_0 is s_0, and that obtained using clauses C is s_1, then the resuting substitution sequence is s_0 *\$sometimes* s_1. More formally,

$$\mathbf{D}[[c_0 :: C]]\rho = \lambda x \lambda y \lambda z.[(\mathbf{C}[[c_0]]\rho\, x\, y\, z)\ \$sometimes\ (\mathbf{D}[[C]]\rho\, x\, y\, z)]$$

where the parameters x, y and z are as in the previous case.

Finally, the meaning of a single clause is a function that takes a tuple of terms τ and a finite substitution σ, and returns a sequence of substitutions. The idea is to first rename variables in the clause uniformly, such that none of the new variables used appear in $dom(\sigma)$, and unify $\sigma(\tau)$ with the tuple of terms appearing in the head of the renamed clause; then, evaluate the body of the clause after applying the resulting substitution to it; and finally, return the substitution sequence obtained by projecting each substitution in the sequence obtained from the evaluation of the body on the set of variables $dom(\sigma)$ that the clause was called with. The reason for the projection is that while the ''interesting'' variables here are the elements of $dom(\sigma)$, the elements of the sequence returned by evaluating the body of the clause may substitute for variables not in $dom(\sigma)$. As discussed earlier, this is not desirable, and such extraneous elements should be projected away. The function **C**[[ ]] is therefore defined as follows:

$$\begin{array}{l}
\mathbf{C}[[\mathrm{p}(\bar{T}_0) \text{ :- } G_0]]\rho = \\
\qquad \mathrm{p} \leftarrow \lambda\bar{S}\lambda\sigma.[\mathit{let}\ (\mathrm{p}(\bar{T}_1) \text{ :- } G_1) = \mathrm{rename}((\mathrm{p}(\bar{T}_0) \text{ :- } G_0), \mathit{dom}(\sigma)); \\
\qquad\quad \theta = \mathit{unify}(\sigma(\bar{S}), \bar{T}_1); \\
\quad \mathit{in} \\
\qquad\quad \text{if } \theta = \mathit{fail} \text{ then } \mathit{nil} \text{ else } (\lambda x.x \downarrow \mathit{dom}(\sigma)) \parallel \mathbf{G}[[G_1]]\rho(\theta\circ\sigma :: \mathit{nil})].
\end{array}$$

The function **G** takes a sequence of literals, an environment defining the meanings of various predicates, and a sequence of substitutions, and returns a sequence of substitutions. In curried form, this is

$$\mathbf{G}[[\ ]] : \mathit{Goal} \rightarrow \mathit{Env} \rightarrow \mathbf{FSubst}^{\infty}_{\$bottom} \rightarrow \mathbf{FSubst}^{\infty}_{\$bottom}.$$

The empty goal represents a successful computation, and in this case the substitution stream returned is the input stream:

$$\mathbf{G}[[\mathit{nil}]]\rho\Theta = \Theta.$$

The meaning of a sequence of literals $L :: G$ is the stream of substitutions obtained by first solving L with the input stream of substitutions, and ''piping'' the resulting substitution stream into G. This gives the meaning of the goal $L :: G$ given an environment ρ and substitution σ to be

$$\mathbf{G}[[L :: G]]\rho\Theta = \mathbf{G}[[G]]\rho(\mathbf{L}[[L]]\rho\Theta).$$

The meaning of a single literal $\mathrm{p}(\bar{T})$ in an environment ρ, given a finite substitution σ, is the substitution sequence Θ obtained by calling the corresponding predicate in that environment for each substitution in the input stream, and concatenating the resulting substitution streams in order. This is described by the function **L**[[ ]]:

$$\mathbf{L}[[\mathrm{p}(\bar{T})]]\rho\Theta = (\rho(\mathrm{p})(\bar{T}))\ \$circle\ \Theta.$$

The semantic functions described are summarized in Figure 1.

Let '$\mathit{fix}\, x.f(x)$' denote $\$lub_n f^n(\$bottom)$. The meaning of a program $\$langle P, G\$rangle$, where P is a sequence of clauses and G a goal, is then defined to be $\mathbf{G}[[G]]\rho_0\ (\varepsilon_V :: \mathit{nil})$, where ρ_0 is essentially $\mathit{fix}\ \rho.\mathbf{D}[[P]]\rho$, and $V = \mathit{vars}(G)$. To define the limit environment ρ_0 precisely, it is necessary to distinguish between the bottom substitution sequence $\$bottom$, which denotes nontermination, and the bottom environment $\$bottom_{Env}$, which maps every predicate defined in the program to $\lambda x\lambda y.\$bottom$, and any predicate not defined in the program to $\lambda x\lambda y.\mathit{nil}$. This is because in typical Prolog systems, execution

$\mathbf{D}[[\ ]] : Clause^{*} \rightarrow Env \rightarrow Env.$

(D1.1) $\mathbf{D}[[nil]]\rho = \lambda x \lambda y \lambda z.nil.$
(D1.2) $\mathbf{D}[[c_0 :: C]]\rho = \lambda x \lambda y \lambda z.[(\mathbf{C}[[c_0]]\rho x\ y\ z)\ \$sometimes\ (\mathbf{D}[[C]]\rho x\ y\ z)]$

$\mathbf{C}[[\ ]] : Clause \rightarrow Env \rightarrow Env.$

(C1.1) $\mathbf{C}[[\mathrm{p}(\overline{T}_0) :- G_0]]\rho =$

$$\begin{aligned}
&\mathrm{p} \leftarrow \lambda\overline{S}\lambda\sigma.[let\ (\mathrm{p}(\overline{T}_1) :- G_1) = \mathrm{rename}((\mathrm{p}(\overline{T}_0) :- G_0), dom(\sigma));\\
&\qquad \theta = unify(\sigma(\overline{S}), \overline{T}_1);\\
&\quad in\\
&\qquad \text{if } \theta = fail \text{ then } nil \text{ else } (\lambda x.x \downarrow dom(\sigma)) \parallel \mathbf{G}[[G_1]]\rho(\theta \circ \sigma :: nil)].
\end{aligned}$$

$\mathbf{G}[[\ ]] : Literal^{*} \rightarrow Env \rightarrow \mathbf{FSubst}^{\infty}_{\$bottom} \rightarrow \mathbf{FSubst}^{\infty}_{\$bottom}.$

(G1.1) $\mathbf{G}[[nil]]\rho\Theta = \Theta.$
(G1.2) $\mathbf{G}[[L :: G]]\rho\Theta = \mathbf{G}[[G]]\rho(\mathbf{L}[[L]]\rho\Theta).$

$\mathbf{L}[[\ ]] : Literal \rightarrow Env \rightarrow \mathbf{FSubst}^{\infty} \rightarrow \mathbf{FSubst}^{\infty}.$

(L1.1) $\mathbf{L}[[\mathrm{p}(\overline{T})]]\rho\Theta = (\rho(\mathrm{p})(\overline{T}))\ \$circle\ \Theta.$

Figure 1: The semantic functions **D**[[ ]] and **G**[[ ]] for Cut-free Prolog

fails if it encounters a predicate not defined in the program. Then, the limit environment defined by a program P is defined to be $\rho_0 = \underset{n}{\$lub}\,\mathbf{D}^n[[P]]\$bottom_{Env}$. Notice that since **G** and **D** are both defined by the composition of continuous functions and (in the case of **G**) simple recursion, they are continuous (see [12]), and hence the fixpoint ρ_0 exists.

3.2. Operational Semantics for Cut-free Prolog

The operational semantics is given by an interpreter that repeatedly transforms a state encoding a ‘‘leftmost’’ SLD-tree traversed as follows:

- At any point, select the leftmost atom of the current goal for resolution.
- When resolving a literal, try the clauses in the sequential order of occurrence in the program.
- If unification fails at any point, backtrack to the most recent choice point and resume execution with the next alternative there.

This execution strategy in fact corresponds to a depth-first backtracking search of the leftmost SLD-tree for the program.

A state of the interpreter consists of a runtime stack describing the state of the computation, and the list of clauses comprising the program. The stack is a list of records, each record describing a path from the root of the SLD-tree to some node in the tree. Each such record is a triple $\mathit{\$langle\, FrameList}, \mathit{Subst}, \mathit{Clauses\$rangle}$, where *FrameList* is a sequence of *Frames*, and each *Frame* represents a goal to be solved; *Subst* is the current (finite) substitution, being developed incrementally; and *Clauses* is a tail segment of the program, consisting of those clauses that are yet to be tried in solving the leftmost literal of the corresponding goal. A *Frame* is a pair consisting of a goal (which is either the user’s query or the right hand side of some clause used to solve the query) and a set of variables Var_p, which is the set of variables the parent of that goal is interested in. Any substitution resulting from the computation of the goal is projected on the variables in Var_p before being returned.

$$\mathit{Frame} ::= \mathit{\$langle\, Atom}^*, \mathit{Var}_p\mathit{\$rangle}$$
$$\mathit{FrameList} ::= \mathit{nil} \mid \mathit{Frame} :: \mathit{FrameList}$$
$$\mathit{Stack} ::= \mathit{nil} \mid \mathit{\$langle\, FrameList}, \mathit{Subst}, \mathit{Clause}^*\mathit{\$rangle} :: \mathit{Stack}$$
$$\mathit{State} ::= \mathit{Stack}.$$

The interpreter is defined by the function *interp*, which maps a state to a (possibly infinite) sequence of substitutions. The interpreter carries around the sequence of clauses that define the predicates in the program:

$$\mathit{interp} : \mathit{State} \times \mathit{Clause}^* \rightarrow \mathbf{FSubst}^\infty.$$

Execution terminates when the runtime stack becomes empty:

$$\mathit{interp}(\mathit{nil}, P) = \mathit{nil}.$$

When the goal to be solved in the current frame becomes empty, a solution has been found. The current substitution is therefore returned, and execution backtracks to search for other solutions:

$$\mathit{interp}(\mathit{\$langle\, nil}, \phi, \mathit{C\$rangle} :: \mathit{St}, P) = \phi :: \mathit{interp}(\mathit{St}, P).$$

On the other hand, if there are no more program clauses to match against a (non-empty) goal, execution fails and backtracking takes place:

$$\mathit{interp}(\mathit{\$langle\, F}_0, \phi, \mathit{nil\$rangle} :: \mathit{St}, P) = \mathit{interp}(\mathit{St}, P), \text{ if } F_0 \neq \mathit{nil}.$$

Given a non-empty goal and a non-empty sequence of clauses, the interpreter tries to solve the leftmost literal in the goal with respect to those clauses. This corresponds to making a call to the procedure defined by that predicate. If the head of the first clause, after appropriate renaming of clause variables, unifies with the leftmost literal, the frame list is extended with a frame describing the subgoal consisting of the body of that clause. The original goal, together with the untried clauses, is saved on the stack so that alternative solutions may be found on backtracking.

$$interp(\$langle\$langle\, L :: G, V_P\$rangle :: F_0, \phi, (H_0 :- B_0) :: C\$rangle :: St_0, P) =$$
$$interp(\$langle\, F_2 :: F_1 :: F_0, \theta\circ\phi, P, \$rangle :: St_1, P), \text{ where}$$
$$H_1 :- B_1 = rename((H_0 :- B_0), dom(\phi));$$
$$\theta = unify(\phi(L), H_1) \qquad (\neq fail);$$
$$V_P' = dom(\phi);$$
$$F_2 = \$langle\, B_1, V_P'\$rangle\,;\ F_1 = \$langle\, G, V_P\$rangle\,;$$
$$St_1 = \$langle\$langle\, L :: G, V_P\$rangle :: F_0, \phi, C\$rangle :: St_0.$$

On the other hand, if the head of the first clause does not unify with the leftmost literal, the clause is discarded and the process repeated with the remaining clauses:

$$interp(\$langle\, Framelist, \phi, (H_0 :- B_0) :: C\$rangle :: St_0, P) = interp(\$langle\, Framelist, \phi, C\$rangle :: St_0, P).$$

When the goal in a frame becomes empty (there may still be other goals to be solved in the current forward execution path), the current substitution is appropriately projected and computation continues with this projected substitution and the remaining framelist. This corresponds to a return from a procedure. Since new variables may have been introduced in the computation from which the return is being made, any such variables occurring in the projected substitution must be taken into account for renaming purposes as far as the remaining forward computation is concerned.

$$interp(\$langle\$langle\, nil, V_P\$rangle :: F_0, \phi, C\$rangle :: St, P) = interp(\$langle\, F_0, \phi\downarrow V_P, P\$rangle :: St, P).$$

Evaluation of a goal G with respect to a clause sequence C that defines the relevant predicates is defined by the expression

$$interp(\$langle\$langle\, G, V\$rangle :: nil, \varepsilon_V, P\$rangle :: nil\$rangle\,, P),$$

where $V = vars(G)$ and ε_V is the identity finite substitution with domain V. The interpreter is summarized in Figure 2.

3.3. Equivalence of Denotational and Operational Semantics

In this section we show that the meaning given to predicates, atoms and goals by the abstract interpreter described above is in fact that specified by the semantic functions described earlier. Before proceeding with the proof of equivalence, however, we require some structural lemmas regarding the behavior of the abstract interperter. The interpreter defines a function

$$interp : State \times Clause^* \rightarrow \mathbf{Subst}^\infty.$$

$interp : State \times Clause^{*} \rightarrow \mathbf{Subst}^{\infty}$.

(I1.1) $interp(nil, P) = nil.$

(I1.2) $interp(\$langle\ nil, \phi, C\$rangle\ :: St, P) = \phi :: interp(St, P).$

(I1.3) $interp(\$langle\ F_0 :: F_1, \phi, nil\$rangle\ :: St, P) = interp(St, P).$

(I1.4) $interp(\$langle\$langle\ L :: G, V_{\mathrm{P}}\$rangle\ :: F_0, \phi, (H_0 :\!- B_0) :: C\$rangle\ :: St_0, P) =$

$\quad interp(\$langle\ F_2 :: F_1 :: F_0, \theta \circ \phi, P\ \$rangle\ :: St_1, P)$, where

$\qquad H_1 :\!- B_1 = rename((H_0 :\!- B_0), dom(\phi));$

$\qquad \theta = unify(\ \phi(L), H_1) \qquad (\neq fail);$

$\qquad V_P{}' = dom(\phi);$

$\qquad F_2 = \$langle\ B_1, V_P{}'\$rangle\ ; F_1 = \$langle\ G, V_P\$rangle\ ;$

$\qquad St_1 = \$langle\$langle\ L :: G, V_P\$rangle\ :: F_0, \phi, C\$rangle\ :: St_0.$

(I1.5) $interp(\$langle\$langle\ L :: G, V_{\mathrm{P}}\$rangle\ :: F_0, \phi, (H_0 :\!- B_0) :: C\$rangle\ :: St_0, P) =$

$\quad interp(\$langle\$langle\ L :: G, V_{\mathrm{P}}\$rangle\ :: F_0, \phi, C\$rangle\ :: St_0, P)$, where

$\qquad H_1 :\!- B_1 = rename((H_0 :\!- B_0), dom(\phi))$, and $unify(\ \phi(L), H_1\ ) = fail.$

(I1.6) $interp(\$langle\$langle\ nil, V_P\$rangle\ :: F_0, \phi, C\$rangle\ :: St, P) = interp(\$langle\ F_0, \phi \downarrow V_P, P\ \$rangle\ :: St, P).$

Figure 2: Abstract Interpreter for Cut-free Prolog

Moreover, since it is defined in terms of a set of disjoint cases where for each case, the right hand side of each expression is built out of (*i*) continuous functions, (*ii*) the symbol *interp*, and (*iii*) composition of symbols from (*i*) and (*ii*), *interp* defines a continuous functional, and we can use fixpoint induction to reason about its least fixpoint.

The first lemma states that given an interpreter stack $F :: St$, the component F encodes the current forward computation, while the remainder of the stack, St, encodes the remaining portion of the SLD-tree that has to be searched:

Lemma 3.1: For any stack frame F and stack St,

$$interp(F :: St, P) \equiv interp(F :: nil, P)\ \$sometimes\ \ interp(St, P).$$

Proof: By fixpoint induction on *interp*. *\$always*

The next two lemmas state that the interpreter carries out a tuple at a time computation, solving the literals in a goal in their left to right order. First, we show that the tuple at a time computation proceeds with the frames in the current forward computation component of the stack being processed in their LIFO order:

Lemma 3.2: For any frame F, framelist F_1, substitution ϕ, and program P and clause list C,

$$\mathit{interp}(\$\mathit{langle}\, F :: F_1, \phi, C\$\mathit{rangle} :: \mathit{nil}, P) \equiv$$
$$[\lambda\theta.\mathit{interp}(\$\mathit{langle}\, F_1, \theta, P\$\mathit{rangle} :: \mathit{nil}, P)]\ \$\mathit{circle}\ \mathit{interp}(\$\mathit{langle}\, F :: \mathit{nil}, \phi, C\$\mathit{rangle} :: \mathit{nil}, P).$$

Proof: By fixpoint induction on *interp*. *\$always*

The next lemma states that computation of the literals within a frame proceds a tuple at a time, in their left to right order:

Lemma 3.3: For any goal $L :: G$, set of variables V_p, framelist F, substitution ϕ, clause list C, and program P,

$$\mathit{interp}(\$\mathit{langle}\$\mathit{langle}\, L :: G, V_p\$\mathit{rangle} :: F, \phi, C\$\mathit{rangle} :: \mathit{nil}, P) \equiv$$
$$(\lambda\theta.\mathit{interp}(\$\mathit{langle}\$\mathit{langle}\, G, V_p\$\mathit{rangle} :: F, \theta, P\$\mathit{rangle} :: \mathit{nil}, P))\ \$\mathit{circle}\ \mathit{interp}(\$\mathit{langle}\$\mathit{langle}\, L :: \mathit{nil}, \mathit{dom}(\phi)\$\mathit{rangle} :: \mathit{nil}, \phi, C\$\mathit{rangle} :: \mathit{nil}, P).$$

Proof : By fixpoint induction on *interp*. *\$always*

Our final lemma concerns the projection away of extraneous substitutions at the return from a call in the interpreter. The lemma states that this can be done in two steps. This is a purely technical lemma necessary for the proof of Theorem 3.1.

Lemma 3.4: For any goal G, substitutions θ and ϕ, stack component St and program P with tail C,

$$\mathit{interp}(\$\mathit{langle}\$\mathit{langle}\, G, \mathit{dom}(\phi)\$\mathit{rangle} :: \mathit{nil}, \theta\circ\phi, C\$\mathit{rangle} :: St, P) \equiv$$
$$(\lambda\sigma.\sigma\downarrow \mathit{dom}(\phi))\ \|\ \mathit{interp}(\$\mathit{langle}\$\mathit{langle}\, G, \mathit{dom}(\theta\circ\phi)\$\mathit{rangle} :: \mathit{nil}, \theta\circ\phi, C\$\mathit{rangle} :: St, P)$$

Proof: By simplification using the rules in Figure 2, and observing that $\mathit{dom}(\phi) \subseteq \mathit{dom}(\theta\circ\phi)$. *\$always*

Finally, consider the effect of failures on substitution sequences. If a goal fails, it returns the empty substitution sequence. As mentioned in the previous section, given a goal $L :: G$, if the literal L fails, then the entire goal fails. This is expressed by the following lemma:

Lemma 3.5: For any goal G and environment ρ, $\mathbf{G}[[G]]\ \rho\ \mathit{nil} = \mathit{nil}$.

Proof: By structural induction on G. $always

We are now in a position to prove the main result of this section:

Theorem 3.1: For any goal G, literal L, program P with tail C, and substitution stream Θ,

$\mathbf{G}[[G]]\,\rho_0\,(\mathbf{L}[[L]]\,\mathbf{D}[[C]]\,\rho_0\,\Theta) \equiv \lambda\sigma.interp($langle$langle\,L :: G, dom(\sigma)$rangle :: nil, \sigma, C$rangle :: nil, P)\ $circle\ \Theta.$

Proof: By fixpoint induction on **G**[[ ]], **D**[[ ]] and *interp*. $always

Corollary (Equivalence of Denotational and Operational Semantics): For any goal G and program P, and substitution sequence Θ,

$$\mathbf{G}[[G]]\rho_0\,\Theta \equiv \lambda\sigma.interp($langle$langle\,G, dom(\sigma)$rangle :: nil, \sigma, P$rangle :: nil, P)\ $circle\ \Theta.$$

4. Semantics of Prolog with Cut

4.1. The ‘‘Cut’’ Construct

One problem that can arise with the simple control strategy of the Prolog interpreter given earlier is that execution may backtrack exhaustively through subtrees of the search tree that cannot contribute to a solution (in extreme cases, exhaustive search through an infinite tree can lead to nontermination of logically correct programs). The *cut* construct returns some control over this backtracking behavior to the user.

Operationally, the effect of a cut is to discard certain backtrack points, so that execution can never backtrack into them. The behavior of cut is not universally agreed upon in all contexts [9]. In practice, however, cuts are most frequently encountered in one of two static contexts: as part of the top-level conjunction in a clause, or within a disjunction in a clause, i.e. either in a context

```
p :– . . .
p :– . . ., !, . . .
p :– . . .
```

or in a context

```
p :– . . .
p :– . . ., ( ( . . . !, . . . ) ;
              ( . . . )
              ), . . .
p :– . . .
```

Most current implementations of Prolog behave similarly in their treatment of cut in these contexts. The expected behavior here is that the backtrack points discarded by a cut will consist of: all those set up by literals to the left of the cut all the way to the beginning of the clause; and the backtrack point for the parent predicate whose definition includes the clause containing the cut, i.e. all remaining alternative clauses for this predicate. Cuts exhibiting this behavior are sometimes referred to as *hard* cuts; this is to distinguish them from cuts which discard the backtrack points set up by literals to the left of the cut in the clause but not the alternative clauses for the predicate, these being referred to as *soft* cuts. We will restrict our attention to cuts that occur statically in the above contexts, and assume them to be hard.

4.2. Denotational Semantics for Prolog with Cut

Since the effect of a cut is to modify the ‘‘rest of the computation’’, it is naturally modelled using a continuation semantics. As before, each literal in the body of a clause acts as a transformer on streams of substitutions: each literal receives a stream of substitutions from those on its left, and in turn feeds a stream of substitutions to those on its right. The action of a cut fits smoothly into this picture: the cut simply discards all but the first substitution from the sequence received from its left brothers. However, this is not enough for modelling the effect of a cut with respect to the clauses that follow. To model the notion of ‘‘remaining clauses’’, we introduce *declaration continuations, DCont*.

A declaration continuation is similar to an environment in that given a predicate name, it returns a function which, when called with a tuple of terms and a substitution, returns a sequence of substitutions. It differs from the environment in which a goal is evaluated in that while the environment reflects the meaning of the ‘‘current’’ list of clauses to solve the goal with, the declaration continuation gives the meaning of the ‘‘remaining’’ clauses, i.e. the alternatives that would be tried were execution to backtrack from the first clause of the list.

Each literal now yields a boolean ‘‘cut flag’’: the value of the cut flag for a goal indicates whether or not a cut was encountered in that goal. The value of the flag is unaffected by non-cut literals, while a cut sets the flag to true. As before, semantic functions **D**[[ ]] and **G**[[ ]] give the meanings of clause and literal sequences respectively, with the auxiliary functions **C**[[ ]] and **L**[[ ]] giving the meanings of individual clauses and literals; they are now extended to incorporate cut flags.

The function **D**[[ ]] gives the meaning of a clause or a sequence of clauses in a given environment and continuation. Given the empty program, the resulting environment is the empty one (to be precise, the function which maps every predicate name to the function that returns the empty substitution sequence for any input). Given a clause sequence $c_0 :: C$, an environment ρ and continuation δ, we first find the meaning ρ_1 of C in the environment ρ and continuation δ. The environment ρ_1 thus represents the solutions that would be found for any predicate p were we to use only the clauses C to solve for p, using environment ρ to solve for literals in the bodies of clauses in C. Thus, ρ_1 represents the ‘‘rest of the program’’ as far as c_0 is concerned:

$$\mathbf{D}[[c_0 :: C]]\$langle\, \rho, \delta\$rangle\; = \mathbf{C}[[c_0]]\; \$langle\, \rho, (\mathbf{D}[[C]]\$langle\, \rho, \delta\$rangle\,)\$rangle\,.$$

The meaning of a single clause given an environment ρ and a continuation δ is obtained as a function which takes, as arguments, a tuple of terms $\bar{t}$ and a finite substitution σ. It first renames variables in the clause with respect to σ, then tries to unify $\sigma(\bar{t})$ with its head. If unification succeeds, it tries to evaluate the body of the clause in environment ρ, passing in the continuation δ. The result of evaluating the body is a substitution sequence Φ and a cut flag. If the value of the cut flag is **f**, indicating that no cuts were encountered, in the clause, then the continuation δ is evaluated with the input tuple of terms and substitution, thereby computing the substitutions resulting from the remaining clauses for that predicate. These substitutions are appended to the sequence Φ to give the full sequence of substitutions. On the other hand, if the value of the cut flag **t**, then a cut must have been encountered in the clause body, so the continuation is not activated. Thus, we have

$$
\begin{array}{l}
\mathbf{C}[[p(\overline{T}) :- B_0]]\mathit{\$langle}\,\rho, \delta\mathit{\$rangle} = \rho[p \leftarrow f], \text{ where} \\
\qquad f = \lambda\overline{S}\lambda\sigma.[\text{let } (p(\overline{T}_1) :- B_1) = \mathit{rename}(\,(p(\overline{T}_0) :- B_0), \mathit{dom}(\sigma)); \\
\qquad\qquad\qquad \theta = \mathit{unify}(\sigma(\overline{S}), \overline{T}_1); \\
\qquad\qquad\quad \text{in} \\
\qquad\qquad\qquad \text{if } \theta = \mathit{fail} \text{ then } \delta(p)\overline{S}\,\sigma \\
\qquad\qquad\qquad \text{else } ((\lambda x.x \downarrow \mathit{dom}(\sigma)) \parallel \Phi)\ \mathit{\$sometimes}\ (\text{if } \mathit{cflag} \text{ then } \mathit{nil} \text{ else } \delta(p)\overline{S}\,\sigma), \\
\qquad\qquad\qquad\qquad \text{where } \mathit{\$langle}\,\Phi, \mathit{cflag}\mathit{\$rangle} = \mathbf{G}[[B_1]]\rho\ \mathit{\$langle}\,(\theta^\circ\sigma :: \mathit{nil}), \mathbf{f}\mathit{\$rangle}\,.
\end{array}
$$

The semantic function **G**[[ ]] gives the meaning of a goal. It takes as arguments an environment in which to look up predicate meanings for the literals in its body; and a pair consisting of a substitution sequence (corresponding to the substitutions generated by literals to the left of the goal being evaluated) and a boolean flag, the ‘‘cut flag’’. It returns a pair consisting of a substitution sequence (those generated by the literals to its left and the goal itself) and a boolean, which has the value **t** if a cut has been encountered in the goal, **f** otherwise.

In any environment, the empty goal signals a successful computation, and the substitutions and continuation passed in from the left are returned unchanged. To solve a compound goal $L :: G$ in an environment ρ, given the input substitution stream Θ and cut flag *cflag*, we first solve L in ρ with Θ and cut flag, and use the resulting substitution stream and cut flag to solve G in environment ρ. Both these cases involve only minor variations of the corresponding cases in the semantics of Prolog without cut given earlier (see Figure 1).

An atom is evaluated by extracting the meaning of the corresponding predicate symnbol from the environment and applying it to the input term and elements of the input substitution stream. In this case, the ‘‘remaining clauses’’ are not affected in any way, so the input cut flag is returned unchanged:

$$\mathbf{L}[[\mathrm{p}(\overline{T})]]\rho\ \mathit{\$langle}\,\Theta, \mathit{cflag}\mathit{\$rangle} = \mathit{\$langle}\,\Phi, \mathit{cflag}\mathit{\$rangle}, \text{ where } \Phi = \rho(\mathrm{p})(\overline{T})\ \mathit{\$circle}\ \Theta.$$

On the other hand, if the literal is a cut, there are two possibilities: if the input substitution stream is empty, then execution must have failed for some literal to the left of the cut, so that the cut is not executed and therefore has no effect. In this case, clearly, execution would have continued by trying the remaining clauses. Thus, the empty substitution sequence is returned, together with the input cut flag. The cut becomes meaningful only if the input substitution sequence is nonempty. In this case, the output substitution sequence is the singleton list consisting of the first element of the input stream. Also, since the remaining clauses have to be discarded, the output cut flag value is **t**:

$$\mathbf{L}[[\,!\,]]\rho\ \mathit{\$langle}\,\Theta, \mathit{cflag}\mathit{\$rangle} = \text{if } \Theta = \mathit{nil} \text{ then } \mathit{\$langle}\,\mathit{nil}, \mathit{cflag}\mathit{\$rangle} \text{ else } \mathit{\$langle}\,\mathit{head}(\Theta), \mathbf{t}\mathit{\$rangle}$$

where the function *head* gives the first element of a nonempty sequence.

The limit environment ρ_0 is now defined as $\mathit{fix}\,\rho.\mathbf{D}[[P]]\mathit{\$langle}\,\rho,\ \mathit{ncont\$rangle}$, where $\mathit{ncont} = \lambda x \lambda y \lambda z.\mathit{nil}$ is the null continuation. The semantic functions are summarized in Figure 3.

4.3. Operational Semantics for Prolog with Cut

The interpreter has to be modified slightly to deal with cut. Since the effect of a cut is to discard certain choice points, and choice points are maintained on the stack, the interpreter now maintains pointers into the stack, at each level, to the point to cut back (i.e. discard choice points) to if a cut is encountered. The *Frame* component is therefore extended with an additional *Stack* element, which we will refer to as the *dump*.

$\mathit{Frame} ::= \mathit{\$langle}\,\mathit{Atom}^*,\ V_P,\ \mathit{Stack\$rangle}\,.$

$\mathit{FrameList} ::= \mathit{nil}\ \mathbf{|}\ \mathit{Frame} :: \mathit{FrameList}.$

$\mathit{Stack} ::= \mathit{nil}\ \mathbf{|}\ \mathit{\$langle}\,\mathit{FrameList},\ \mathit{Subst},\ \mathit{Clause}^*\mathit{\$rangle}\ :: \mathit{Stack}.$

The actions of the interpreter also have to be modified slightly. There are two main points to note. First, observe that for a cut encountered when evaluating the body of a clause, only those choice points set up *during* and *after* the call that led to that clause will be discarded. Therefore in any invocation of a predicate, it is enough to pass the stack component just before the call is made, into the call as the dump component of the callee. The second is that when a cut is encountered, the backtrack stack should be set to the current dump component, thereby resulting in some choice points at the top of the stack being discarded. The resulting interpreter is summarized in Figure 4.

4.4. Equivalence of Denotational and Operational Semantics

In this section, we show that the meaning given to programs by the abstract interpreter defined above, in the case of Prolog with cut, is the same as that given by the semantic functions.

Unfortunately, unlike the cut-free case, the non-local effects of cut preclude simple structural decompositions of the interpreter state as in Lemmas 3.1-3.5, and make the proof more complex. The main problem is that once cuts are introduced, some of the substitutions generated while evaluating a goal may have to be discarded if a cut was encountered during the execution of that goal; however, *interp* does not indicate whether or not a cut was encountered during the execution of a goal. Moreover, the interpreter only works on one substitution at a time, and encodes the other computation paths in a rather complicated manner in its stack, the effect of cut being to discard certain backtrack points at the top of the stack. This makes it difficult to find simple decomposition lemmas for *interp* in the presence of cuts. Instead of a direct proof of equivalence, therefore, we resort to an intermediate function, *tran*, that mediates between the denotational and the operational semantics. This is very similar to *interp*, except that *tran* works on substitution sequences and uses a boolean flag to indicate whether a cut was encountered, much like the semantic functions of Figure 3. This function is illustrated in Figure 5.

Top level goals are evaluated through the auxiliary semantic function $\hat{\mathbf{G}}$, which is similar to $\mathbf{G}[[\ ]]$ except that it takes as argument two environments – one to evaluate the leftmost literal of the goal in, and

$$\mathit{DCont} = \mathit{Predicate} \rightarrow \mathit{Term} \rightarrow \mathbf{FSubst}_{\text{\$bottom}} \rightarrow \mathbf{FSubst}^{\infty}_{\text{\$bottom}}.$$

$\mathbf{D}[[\ ]] : \mathit{Clause}^{*} \rightarrow (\mathit{Env} \times \mathit{DCont}) \rightarrow \mathit{Env}$

(D2.1) $\mathbf{D}[[\mathit{nil}]]\text{\$langle}\,\rho, \delta\text{\$rangle} = \lambda x \lambda y \lambda z.\mathit{nil}.$

(D2.2) $\mathbf{D}[[c_0 :: C]]\text{\$langle}\,\rho, \delta\text{\$rangle} = \mathbf{C}[[c_0]]\ \text{\$langle}\,\rho, (\mathbf{D}[[C]]\text{\$langle}\,\rho, \delta\text{\$rangle})\text{\$rangle}.$

$\mathbf{C}[[\ ]] : \mathit{Clause} \rightarrow (\mathit{Env} \times \mathit{DCont}) \rightarrow \mathit{Env}$

(C2.1) $\mathbf{C}[[p(\bar{T}) :- B_0]]\text{\$langle}\,\rho, \delta\text{\$rangle} = \rho[p \leftarrow f]$, where

$f = \lambda \bar{S} \lambda \sigma.[\text{let } (p(\bar{T}_1) :- B_1) = \mathit{rename}(\ (p(\bar{T}_0) :- B_0), \mathit{dom}(\sigma));$

$\theta = \mathit{unify}(\sigma(\bar{S}), \bar{T}_1);$

in

$\text{if } \theta = \mathit{fail} \text{ then } (\delta(p)\bar{Y})(\sigma)$

$\text{else } ([\lambda x.x \downarrow \mathit{dom}(\sigma)] \parallel \Phi)\ \text{\$sometimes}\ (\text{if } \mathit{cflag} \text{ then } \mathit{nil} \text{ else } \delta(p)\bar{Y})(\sigma)),$

$\text{where } \text{\$langle}\,\Phi, \mathit{cflag}\text{\$rangle} = \mathbf{G}[[B_1]]\rho\ \text{\$langle}\,(\theta^{\circ}\sigma :: \mathit{nil}), \mathbf{f}\text{\$rangle}\,].$

$\mathbf{G}[[\ ]] : \mathit{Literal}^{*} \rightarrow \mathit{Env} \rightarrow (\mathbf{FSubst}^{\infty}_{\text{\$bottom}} \times \mathbf{Bool}) \rightarrow (\mathbf{FSubst}^{\infty}_{\text{\$bottom}} \times \mathbf{Bool})$

(G2.1) $\mathbf{G}[[\mathit{nil}]]\rho\ \text{\$langle}\,\Theta, \mathit{cflag}\text{\$rangle} = \text{\$langle}\,\Theta, \mathit{cflag}\text{\$rangle}.$

(G2.2) $\mathbf{G}[[L :: G]]\rho\ \text{\$langle}\,\Theta, \mathit{cflag}\text{\$rangle} = \mathbf{G}[[G]]\rho\ (\mathbf{L}[[L]]\rho\ \text{\$langle}\,\Theta, \mathit{cflag}\text{\$rangle}.$

$\mathbf{L}[[\ ]] : \mathit{Literal} \rightarrow \mathit{Env} \rightarrow (\mathbf{FSubst}^{\infty}_{\text{\$bottom}} \times \mathbf{Bool}) \rightarrow (\mathbf{FSubst}^{\infty}_{\text{\$bottom}} \times \mathbf{Bool})$

(L2.1) $\mathbf{L}[[\,!\,]]\rho\ \text{\$langle}\,\Theta, \mathit{cflag}\text{\$rangle} = \text{if } \Theta = \mathit{nil} \text{ then } \text{\$langle}\,\mathit{nil}, \mathit{cflag}\text{\$rangle} \text{ else } \text{\$langle}\,\mathit{head}(\Theta), \mathbf{t}\text{\$rangle}.$

(L2.2) $\mathbf{L}[[\mathrm{p}(\bar{T})]]\rho\ \text{\$langle}\,\Theta, \mathit{cflag}\text{\$rangle} = \text{\$langle}\,\Phi, \mathit{cflag}\text{\$rangle}$, where $\Phi = (\rho(\mathrm{p})(\bar{T}))\ \text{\$circle}\ \Theta.$

Figure 3: Semantic Functions for Prolog with Cut

(I2.1) $\mathit{interp}(\mathit{nil}, P) = \mathit{nil}$.

(I2.2) $\mathit{interp}(\langle \mathit{nil}, \phi, C\rangle :: \mathit{St}, P) = \phi :: \mathit{interp}(\mathit{St}, P)$.

(I2.3) $\mathit{interp}(\langle F_0 :: \mathit{FRest}, \phi, \mathit{nil}\rangle :: \mathit{St}, P) = \mathit{interp}(\mathit{St}, P)$.

(I2.4) $\mathit{interp}(\langle\langle\, ! :: A, V_{\mathrm{p}}, D\rangle :: F_0, \phi, C\rangle :: \mathit{St}, P) = \mathit{interp}(\langle\langle A, V_{\mathrm{p}}, D\rangle :: F_0, \phi, P\rangle :: D, P)$.

(I2.5) $\mathit{interp}(\langle\langle L :: G, V_{\mathrm{p}}, D\rangle :: F_0, \phi, (H_0 :\!- B_0) :: C\rangle :: \mathit{St}_0, P)\;\; (L \neq \text{‘!’}) =$

$\mathit{interp}(\langle F_2 :: F_1 :: F_0, \theta\circ\phi, P\rangle :: \mathit{St}_1, P)$, where

$H_1 :\!- B_1 = \mathit{rename}((H_0 :\!- B_0), \mathit{dom}(\phi))$;

$\theta = \mathit{unify}(\phi(L), H_1)$ $(\neq \mathit{fail})$;

$V_P{}' = \mathit{dom}(\phi)$;

$F_2 = \langle B_1, V_P{}', \mathit{St}_0\rangle$; $F_1 = \langle G, V_{\mathrm{p}}, D\rangle$;

$\mathit{St}_1 = \langle\langle L :: G, V_{\mathrm{p}}, D\rangle :: F_0, \phi, C\rangle :: \mathit{St}_0$.

(I2.6) $\mathit{interp}(\langle\langle L :: G, V_{\mathrm{p}}, D\rangle :: F_0, \phi, (H_0 :\!- B_0) :: C\rangle :: \mathit{St}_0, P)\;\; (L \neq \text{‘!’}) =$

$\mathit{interp}(\langle\langle L :: G, V_{\mathrm{p}}, D\rangle :: F_0, \phi, C\rangle :: \mathit{St}_0, P)$, where

$H_1 :\!- B_1 = \mathit{rename}((H_0 :\!- B_0), \mathit{dom}(\phi))$, and $\mathit{unify}(\phi(L), H_1) = \mathit{fail}$.

(I2.7) $\mathit{interp}(\langle\langle \mathit{nil}, V_P, D\rangle :: F_0, \phi, C\rangle :: \mathit{St}, P) = \mathit{interp}(\langle F_0, \phi\downarrow V_P, P\rangle :: \mathit{St}, P)$.

Figure 4 : Abstract Interpreter for Prolog with Cut

(T2.1) $\mathit{tran}(\mathit{nil}, P) = \mathit{nil}$.

(T2.2) $\mathit{tran}(\mathit{\$langle}\, \mathit{nil}, \Phi, C\mathit{\$rangle} :: St, P) = \Phi\ \mathit{\$sometimes}\ \mathit{tran}(St, P)$.

(T2.3) $\mathit{tran}(\mathit{\$langle}\, F :: \mathit{FRest}, \Phi, \mathit{nil\$rangle} :: St, P) = \mathit{tran}(St, P)$.

(T2.4) $\mathit{tran}(\mathit{\$langle\$langle}\, G_1\ \mathit{\$sometimes}\ G_2, V_P, D\mathit{\$rangle} :: \mathit{FRest}, \Psi, C\mathit{\$rangle} :: St, P) =$

let $\mathit{\$langle}\, \Phi, \mathit{cflag\$rangle} = \hat{\mathbf{G}}[[G_1]]\rho_0\ (\mathbf{D}[[C]]\mathit{\$langle}\, \rho_0, \mathit{ncont\$rangle})\ \mathit{\$langle}\, \Psi, \mathbf{f}\mathit{\$rangle}$ in

if $\Phi = \mathit{nil}$ then

$\mathit{tran}($ (if cflag then D else St), P);

if $\Phi \neq \mathit{nil}$ then

$\mathit{tran}(\mathit{\$langle\$langle}\, G_2, V_P, D\mathit{\$rangle} :: \mathit{FRest}, \Phi, P\mathit{\$rangle} ::$ (if cflag then D else St), P).

(T2.5) $\mathit{tran}(\mathit{\$langle\$langle}\, \mathit{nil}, V_P, D\mathit{\$rangle} :: \mathit{FRest}, \Phi, C\mathit{\$rangle} :: St, P) =$

$\mathit{tran}(\mathit{\$langle}\, \mathit{FRest}, (\lambda x.x \downarrow V_P) \parallel \Phi, P\mathit{\$rangle} :: St, P)$.

Figure 5

the other to evaluate the rest of the goal in.

$$\hat{\mathbf{G}}[[\ ]] : \mathit{Literal}^* \to \mathit{Env} \to \mathit{Env} \to (\mathbf{FSubst}^{\infty}_{\mathit{\$bottom}} \times \mathbf{Bool}) \to (\mathbf{FSubst}^{\infty}_{\mathit{\$bottom}} \times \mathbf{Bool})$$

$$\hat{\mathbf{G}}[[L :: G]]\rho_1\ \rho_2\ \mathit{\$langle}\, \Phi, b\mathit{\$rangle} = \mathbf{G}[[G]]\rho_1\ (\mathbf{L}[[L]]\rho_2\ \mathit{\$langle}\, \Phi, b\mathit{\$rangle}).$$

$$\hat{\mathbf{G}}[[\mathit{nil}]]\rho_1\ \rho_2\ \mathit{\$langle}\, \Phi, b\mathit{\$rangle} = \mathbf{G}[[\mathit{nil}]]\rho_2\ \mathit{\$langle}\, \Phi, b\mathit{\$rangle}.$$

As before, we state some structural lemmas regarding $\mathbf{G}[[\ ]]$ and *tran*. If a sequence A_1 is a prefix of a sequence A_2, we will write $A_1 \le A_2$. It is easy to show that $\mathbf{G}[[\ ]]$ is monotonic with respect to $\le$:

Lemma 4.1: If $\Phi_1 \le \Phi_2$, $\mathit{\$langle}\, \Psi_1, \mathit{cf}_1\mathit{\$rangle} = \mathbf{G}[[G]]\rho\ \mathit{\$langle}\, \Phi_1, \mathit{cf}_0\mathit{\$rangle}$, and $\mathit{\$langle}\, \Psi_2, \mathit{cf}_2\mathit{\$rangle} = \mathbf{G}[[G]]\rho\ \mathit{\$langle}\, \Phi_2, \mathit{cf}_0\mathit{\$rangle}$, then $\Psi_1 \le \Psi_2$.

Proof: By structural induction on G. *\$always*

Lemma 4.2: Let $\mathit{\$langle}\, \Psi_0, \mathit{cf}_0\mathit{\$rangle} = \mathbf{G}[[G]]\rho\ \mathit{\$langle}\, \Phi_1\ \mathit{\$sometimes}\ \Phi_2, \mathbf{f}\mathit{\$rangle}$, $\mathit{\$langle}\, \Psi_1, \mathit{cf}_1\mathit{\$rangle} = \mathbf{G}[[G]]\rho\ \mathit{\$langle}\, \Phi_1, \mathbf{f}\mathit{\$rangle}$, and $\mathit{\$langle}\, \Psi_2, \mathit{cf}_2\mathit{\$rangle} = \mathbf{G}[[G]]\rho\ \mathit{\$langle}\, \Phi_2, \mathbf{f}\mathit{\$rangle}$. Then, if cf_1 is false, then $\Psi_0 = \Psi_1\ \mathit{\$sometimes}\ \Psi_2$.

Proof: By structural induction on G. *\$always*

Lemma 4.3: Let $\textit{\$langle}\,\Psi_0, cf_0\textit{\$rangle} = \mathbf{G}[[G]]\rho\ \textit{\$langle}\,\Phi_1\ \textit{\$sometimes}\ \Phi_2, \mathbf{f}\textit{\$rangle}$, and $\textit{\$langle}\,\Psi_1, cf_1\textit{\$rangle} = \mathbf{G}[[G]]\rho\ \textit{\$langle}\,\Phi_1, \mathbf{f}\textit{\$rangle}$. Then, if cf_1 is true and $\Psi_1 \neq nil$, then $\Psi_0 = \Psi_1$.

Proof: By structural induction on G. *$always*

Lemma 4.4: For any framelist F, substitution sequence $\Phi_1\ \textit{\$sometimes}\ \Phi_2$, stack component St and program P,

$$tran(\textit{\$langle}\,F, \Phi_1\ \textit{\$sometimes}\ \Phi_2, P\textit{\$rangle} :: St, P) = tran(\textit{\$langle}\,F, \Phi_1, P\textit{\$rangle} :: \textit{\$langle}\,F, \Phi_2, P\textit{\$rangle} :: St, P).$$

Proof: By structural induction on F (see Appendix 2). *$always*

The equivalence of the denotational and operational semantics is then asserted by the following theorem:

Theorem 4.1: For any framelist F, substitution θ, clause list C, stack component St and program P,

$$interp(\textit{\$langle}\,F, \theta, C\textit{\$rangle} :: St, P) \equiv tran(\textit{\$langle}\,F, \theta :: nil, C\textit{\$rangle} :: St, P).$$

Proof: By fixpoint induction on *interp*, *tran*, **G**[[ ]] and **D**[[ ]]. *$always*

Corollary (Equivalence of Denotational and Operational Semantics): If

$\Phi = interp(\textit{\$langle\$langle}\,G, dom(\theta), nil\textit{\$rangle} :: nil, \theta, P\textit{\$rangle} :: nil, P)$, then
$\mathbf{G}[[G]]\rho_0\ \textit{\$langle}\,\theta :: nil, \mathbf{f}\textit{\$rangle} = \textit{\$langle}\,\Phi, ncont\textit{\$rangle}$.

where $\rho_0 = \mathit{fix}\ \rho.\ \mathbf{D}[[P]]\textit{\$langle}\,\rho, ncont\textit{\$rangle}$.

5. Applications

A central motivation for developing a denotational semantics for Prolog has been the need to justify the correctness of transformations on Prolog programs. Typically, such justification is useful for validating transformations used in optimizing compilers. While there are plenty of ''folk theorems'', such as that contiguous cuts are idempotent, i.e. 'L_1, !, !, L_2' ≡ 'L_1, !, L_2', none are firmly based on a semantics that describes the computational behavior of programs. The need for a denotational formulation of the semantics follows from its ability to support reasoning about the *strong* correctness of transformations. This is particularly important in the case of Prolog, since many transformations involve the insertion of cuts [2, 4, 8, 11], which may change the termination behavior of programs.

As an example, we prove the correctness of two transformations involving the manipulation of cuts. The first involves the removal of cuts in certain contexts, while the second involves the insertion of cuts to constrain the search space. A predicate (clause, literal) is *determinate* in a program if any call to it in that program yields at most one substitution, i.e. the output sequence has length at most 1. By '$P\backslash_q$' we mean a sequence of clauses without any clause defining q.

The first theorem we prove states that if a cut appears as the last literal in the last clause of a predicate, and that clause is determinate independently of the cut, then the cut can be discarded without affecting the semantics of the program.

Theorem 5.1: Let P and Q be the programs

$P = (p(\bar{T}_{11}) :- q_1(\bar{T}_{12}) :: nil)) :: ... :: (p(\bar{T}_{n1}) :- q_n(\bar{T}_{n2}) :: ! :: nil) :: P\backslash_p$, and

$Q = (p(\bar{T}_{11}) :- q_1(\bar{T}_{12}) :: nil)) :: ... :: (p(\bar{T}_{n1}) :- q_n(\bar{T}_{n2}) :: nil) :: P\backslash_p$.

Then, if $q_n(\bar{T}_{n2})$ is determinate in both P and Q, then $\rho_P \equiv \rho_Q$, where $\rho_P = fix\ \rho.\mathbf{D}[[P]]\$langle\ \rho,$ $ncont\$rangle$ and $\rho_Q = fix\ \rho.\mathbf{D}[[Q]]\$langle\ \rho,\ ncont\$rangle$.

Proof Outline: The proof is by fixpoint induction on ρ_P and ρ_Q, showing that for any tuple of terms $\bar{T}$ and substitution θ,

$$\rho_P(p)\ \bar{T}\ \theta \equiv \rho_Q(p)\ \bar{T}\ \theta.$$

For the interesting case, we need to show that

$\mathbf{C}[[p(\bar{T}_{n1}) :- q_n(\bar{T}_{n2}) :: ! :: nil]]\$langle\ \rho_P,\ \delta\$rangle\ \ p\ \bar{T}\ \theta \equiv \mathbf{C}[[p(\bar{T}_{n1}) :- q_n(\bar{T}_{n2}) :: nil]]\$langle\ \rho_Q,\ \delta'\$rangle\ \ p\ \bar{T}\ \theta.$

where $\delta = \mathbf{D}[[P_p]]\$langle\ \rho_P,\ ncont\$rangle$, $\delta' = \mathbf{D}[[P_p]]\$langle\ \rho_Q,\ ncont\$rangle$. There are two cases that have to be considered:

Case I: Unification of $\bar{T}_{n1}$ and $\theta(\bar{T})$ fails: in this case, the left hand side is

$\delta(p)\ \bar{T}\ \theta \equiv nil \equiv \delta'(p)\ \bar{T}\ \theta$

whence the left and right hand sides are equal.

Case II: Unification succeeds with unifier ϕ: in this case, the left hand side is

$\Theta\ \$sometimes\ \ (\delta_1(p)\ \bar{T}\ \theta)$, where $\$langle\ \Phi,\ \delta_1\$rangle\ = \mathbf{G}[[q_n(\bar{T}_{n2}) :: ! :: nil]]\rho_P\ \$langle\ \phi\circ\theta :: nil,\ \delta\$rangle$.

Observe that q_n is determinate in P and Q, whence it is easy to show that

$\$langle\ \Phi,\ \delta_1\$rangle\ = \mathbf{L}[[q_n(\bar{T}_{n2})]]\rho_P\ \$langle\ \phi\circ\theta :: nil,\ \delta\$rangle$,

and $\delta_1(p) \equiv ncont(p)$, since δ is defined in terms of $P\backslash_p$, which does not contain any clauses for p. Thus, the left hand side can be written as $\mathbf{L}[[q_n(\bar{T}_{n2})]]\rho_P\ \$langle\ \phi\circ\theta :: nil,\ ncont\$rangle$, and from the induction hypothesis, this is equal to $\mathbf{L}[[q_n(\bar{T}_{n2})]]\rho_Q\ \$langle\ \phi\circ\theta :: nil,\ ncont\$rangle$. The right hand side is $\$langle\ \ \Phi\ \$sometimes\ \ \delta'(p)\ \bar{T}\ \theta)$, where

$\$langle\ \Phi,\ \delta_2\$rangle\ = \mathbf{G}[[q(\bar{T}_{n2}) :: nil]]\rho_Q\ \ \$langle\ \phi\circ\theta :: nil,\ \delta'\$rangle$,

where $\delta_2(p) \equiv ncont(p)$ from the definition of δ'. This reduces to $\mathbf{L}[[q_n(\bar{T}_{n2})]]\rho_0'\$langle\ \phi\circ\theta :: nil,$ $ncont\$rangle$, whence the left and right hand sides are equal, and the theorem holds. *\$always*

We have considered the case where the last clause for p has a single literal in its body, but the generalization to a list of literals is obvious. The practical utility of this result is that it makes possible the elimination of some redundant cuts, which in turn leads to improved space utilization because opportunities open up for tail recursion optimization (in general, last goal optimization). For example, the predicate

append1([],L,L).
append1([H|L1],L2,[H|L3]) :– append1(L1,L2,L3), !.

is not tail recursive, and needs linear space to concatenate two lists. On the other hand, the predicate

append2([],L,L).
append2([H|L1],L2,[H|L3]) :– append2(L1,L2,L3).

is tail recursive, and can concatenate two lists using constant space. Both *append1* and *append2* are determinate when called with the first two arguments as ground terms, and from the theorem above, any call to *append1* whose first two arguments are ground can be replaced by a call to *append2* (information regarding the instantiations of arguments in a call can be obtained via mode analysis, see [3]).

The second transformation we validate involves the insertion of cuts in determinate predicates to reduce the amount of search. Similar transformations have been proposed by several researchers [4, 8, 11], but none, to our knowledge, have been formally validated.

Theorem 5.2: Let P and Q be the programs

$P = (p(\bar{T}_{11}) :- q_1(\bar{T}_{12}) :: nil)) :: (p(\bar{T}_{21}) :- q_2(\bar{T}_{22}) :: nil) :: \ldots :: (p(\bar{T}_{n1}) :- q_n(\bar{T}_{n2}) :: nil) :: P\backslash_p,$

$Q = (p(\bar{T}_{11}) :- q_1(\bar{T}_{12})) :: ! :: nil) :: (p(\bar{T}_{21}) :- q_1(\bar{T}_{22}) :: ! :: nil) :: \ldots :: (p(\bar{T}_{n1}) :- q_n(\bar{T}_{n2}) :: nil) :: P\backslash_p.$

Then, if p is determinate in both P and Q, then ρ_P *$le* ρ_Q, where $\rho_P = fix\ \rho.\mathbf{D}[[P]]$*$langle* ρ, *ncont$rangle* and $\rho_Q = fix\ \rho.\mathbf{D}[[Q]]$*$langle* ρ, *ncont$rangle* .

Proof Outline: The proof is by fixpoint induction on ρ_P and ρ_Q, showing that for any tuple of terms $\bar{T}$ and substitution θ,

$\rho_P(p)\ \bar{T}\ \theta$ *$le* $\rho_Q(p)\ \bar{T}\ \theta.$

For the interesting case, we need to show that

$\mathbf{C}[[p(\bar{T}_{11}) :- q_1(\bar{T}_{12}) :: nil]]$*$langle* ρ_P, δ*$rangle* $p\ \bar{T}\ \theta$ *$le* $\mathbf{C}[[p(\bar{T}_{11}) :- q_1(\bar{T}_{12}) :: ! :: nil]]$*$langle* ρ_Q, δ'*$rangle* $p\ \bar{T}\ \theta$

where

$\delta = \mathbf{D}[[(p(\bar{T}_{21}) :- q_2(\bar{T}_{22}) :: nil) :: \ldots :: (p(\bar{T}_{n1}) :- q_n(\bar{T}_{n2}) :: nil) :: P\backslash_p]]$*$langle* ρ_P, *ncont$rangle* ,

and

$\delta' = \mathbf{D}[[(p(\bar{T}_{21}) :- q_1(\bar{T}_{22}) :: nil) :: ! :: nil)) :: \ldots :: (p(\bar{T}_{n1}) :- q_n(\bar{T}_{n2}) :: nil) :: P\backslash_p]]$*$langle* ρ_Q, *ncont$rangle*

given ρ_P *$le* ρ_Q, δ *$le* δ'. There are two cases to be considered:

Case I: Unification of $\bar{T}_{11}$ and $\theta(\bar{T})$ fails. In this case, the left hand side is

$\delta(p)\ \bar{T}\ \theta$ *$le* $\delta'(p)\ \bar{T}\ \theta \equiv$ right hand side

whence the theorem holds.

Case II: Unification succeeds with unifier ϕ. The left hand side is $\Phi\ \textit{\$sometimes}\ (newcont(p)\ \bar{T}\ \theta)$, where $\textit{\$langle}\,\Phi,\ newcont\textit{\$rangle} = \mathbf{G}[[q_1(\bar{T}_{12}') :: nil]]\rho_P\ \textit{\$langle}\,\phi\circ\theta :: nil,\ \delta\textit{\$rangle}$, and $\bar{T}_{12}'$ is the appropriate alphabetic variant of $\bar{T}_{12}$. There are two subcases:

(*a*) $\Phi = nil$. Then, $\textit{\$langle}\,\Phi,\ newcont\textit{\$rangle} = \mathbf{G}[[q_1(\bar{T}_{12}') :: \,!\, :: nil]]\ \rho_P\ \textit{\$langle}\,\phi\circ\theta :: nil,\ \delta\textit{\$rangle}$.

(*b*) $\Phi \neq nil$. Since p is determinate in P, we must have $(newcont(p)\ \bar{T}\ \theta) = \textit{\$bottom}$ or nil, and $\textit{\$langle}\,\Phi,\ ncont\textit{\$rangle} = \mathbf{G}[[q_1(\bar{T}_{12}') :: \,!\, :: nil]]\ \rho_P\ \textit{\$langle}\,\phi\circ\theta :: nil,\ \delta\textit{\$rangle}$.

The right hand side is $\textit{\$langle}\,\Phi,\ newcont'\textit{\$rangle} = \mathbf{G}[[q_1(\bar{T}_{12}') :: \,!\, :: nil]]\rho_Q\ \textit{\$langle}\,\phi\circ\theta :: nil,\ \delta'\textit{\$rangle}$. From the inductive hypothesis and monotonicity, we then have

$$\Phi\ \textit{\$sometimes}\ (newcont(p)\ \bar{T}\ \theta)\ \textit{\$le}\ \Phi'\ \textit{\$sometimes}\ (newcont'(p)\ \bar{T}\ \theta)$$

and the theorem holds. *$always*

Notice that the insertion of cuts can change the termination behavior of programs, so that an otherwise nonterminating program may terminate once cuts have been inserted.

6. Conclusions

The semantics of Prolog has traditionally been given in terms of the model theory of first order logic. However, such a semantics is often inadequate for reasoning about the computational behavior of Prolog programs, and for validating the strong correctness of program transformations, since it does not support reasoning about the computational behavior of such programs. The problem becomes even more serious if we wish to deal with programs that contain ''impure'' features such as *cut*.

In this paper, we gave a denotational and operational semantics for Prolog, both with and without cut; proved the congruence of these semantics; and demonstrated the utility of such a semantics by validating two optimizing transformations on Prolog programs. We believe that while the model theoretic semantics is very useful for understanding Prolog programs, it is necessary to resort to a denotational description such as this in order to reason about tools that manipulate and transform Prolog programs.

References

1. K. R. Apt and M. H. van Emden, Contributions to the Theory of Logic Programming, *J. ACM 29*, 3 (July 1982), pp. 841-862.

2. S. K. Debray, Towards Banishing the Cut from Prolog, in *Proc. 1986 Int. Conf. on Computer Languages*, IEEE Computer Society, Miami Beach, Florida, Oct. 1986, pp. 2-12.

3. S. K. Debray and D. S. Warren, Automatic Mode Inference for Logic Programs, *J. Logic Programming 5*, 3 (Sep. 1988), pp. 207-229.

4. S. K. Debray and D. S. Warren, Functional Computations in Logic Programs, *ACM Transactions on Programming Languages and Systems 11*, 3 (July 1989), pp. 451-481.

5. G. Frandsen, Logic Programming, Substitutions and Finite Computability, DAIMI PB-186, Computer Science Department, Aarhus University, Denmark, Jan. 1985.

6. G. Frandsen, Logic Programming and Substitutions, in *Proc. International Conference on Fundamentals of Computation Theory*, Springer-Verlag, , 146-158. LNCS v. 199.

7. N. D. Jones and A. Mycroft, Stepwise Development of Operational and Denotational Semantics for PROLOG, in *Proc. 1984 Int. Symposium on Logic Programming*, IEEE Computer Society, Atlantic City, New Jersey, Feb. 1984, 289-298.

8. C. S. Mellish, Some Global Optimizations for a Prolog Compiler, *J. Logic Programming 2*, 1 (Apr. 1985), 43-66.

9. C. Moss, Results of Cut Tests, in *Prolog Electronic Digest, Vol. 3, No. 42*, Oct 9, 1985.

10. R. A. O'Keefe, On the Treatment of Cuts in Prolog Source-Level Tools, in *Proc. 1985 Symposium on Logic Programming*, Boston, July 1985, 73-77.

11. H. Sawamura and T. Takeshima, Recursive Unsolvability of Determinacy, Solvable Cases of Determinacy and Their Applications to Prolog Optimization, in *Proc. 1985 Symposium on Logic Programming*, Boston, July 1985, 200-207.

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

13. M. H. van Emden and R. A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *J. ACM 23*, 4 (Oct. 1976), pp. 733-742.

Appendix 1: Proofs of Equivalence for Prolog without Cut

Lemma 3.1: For any goal G and environment ρ, **G**[[G]] ρ *nil* = *nil*.

Proof: By structural induction on G. The base case, G = *nil*, follows from the definition of **G**[[ ]]. In the inductive case, consider a goal $p(\bar{T})$ :: G. From the definition of **G**[[ ]], this is equivalent to **G**[[G]]ρ (**L**[[$p(\bar{T})$]]ρ *nil*), where

L[[L]]ρ *nil*
= (ρ(p) $\bar{T}$) *$circle* *nil*
= *nil* (from the definition of *$circle*).

This gives **G**[[$p(\bar{T})$ :: G]] ρ *nil* = **G**[[G]] ρ *nil*, whence the theorem follows from the inductive hypothesis. *$always*

To simplify the notation in the proof of the next theorem, if an expression E_0 reduces to an expression E_1 by application of rule R, we will write this as

$$E_0 \ \$rarrow_R \ E_1.$$

Thus, for example, if E_0 reduces to E_1 from clause G1.1 of the definition of **G**[[ ]] and clause D1.2 of that of **D**[[ ]], we will write

$$E_0 \ \$rarrow_{(\mathrm{G1.1,\ D1.2})} \ E_1.$$

Where the source of the reduction is obvious, these annotations will sometimes be omitted.

Theorem 3.1: For any goal G, literal L, program P with tail C, and substitution stream Θ,

G[[G]] ρ_0 (**L**[[$p(\bar{T})$]] **D**[[C]] ρ_0 Θ) ≡
λσ.*interp*(*$langle$langle* $p(\bar{T})$ :: G, *dom*(σ)*$rangle* :: *nil*, σ, C*$rangle* :: *nil*, P) *$circle* Θ.

Proof: By fixpoint induction on **G**[[ ]], **D**[[ ]] and *interp*. For the inductice step, there are six relevant cases:

Case 1: G = *nil*, C = *nil*.

Case 2: G = *nil*, $C = c_0 :: C'$, unification of $p(\bar{T})$ with the head of c_0 succeeds.

Case 3: G = *nil*, $C = c_0 :: C'$, unification of $p(\bar{T})$ with the head of c_0 fails.

Case 4: $G \neq$ *nil*, C = *nil*.

Case 5: $G \neq$ *nil*, $C = c_0 :: C'$, unification of $p(\bar{T})$ with the head of c_0 succeeds.

Case 6: $G \neq$ *nil*, $C = c_0 :: C'$, unification of $p(\bar{T})$ with the head of c_0 fails.

The proof proceeds as follows:

Case 1: The left hand side is

$\mathbf{G}[[nil]]\ \rho_0\ (\mathbf{L}[[p(\bar{T})]]\ \mathbf{D}[[nil]]\ \rho_0\ \Theta)$

$\text{\$rarrow}_{\text{(G1.1, D1.1)}}\quad \mathbf{L}[[p(\bar{T})]]\ \lambda x\lambda y\lambda z.nil$

$\text{\$rarrow}_{\text{L1.1}}\quad (\lambda x\lambda y\lambda z.nil\ p\ \bar{T})\ \text{\$circle}\ \Theta$

$\text{\$rarrow}\quad nil.$

The right hand side is

$\lambda\sigma.interp(\text{\$langle\$langle}\, p(\bar{T}) :: G,\ dom(\sigma)\text{\$rangle} :: nil,\ \sigma,\ nil\text{\$rangle} :: nil,\ P)\ \text{\$circle}\ \Theta$

$\text{\$rarrow}_{\text{I1.3}}\quad \lambda\sigma.interp(nil, P)\ \text{\$circle}\ \Theta$

$\text{\$rarrow}_{\text{I1.1}}\quad \lambda\sigma.nil\ \text{\$circle}\ \Theta$

$\text{\$rarrow}\quad nil.$

Case 2: The left hand side is

$\mathbf{G}[[nil]]\ \rho_0\ (\mathbf{L}[[p(\bar{T})]]\ \mathbf{D}[[(H_0 :- B_0) :: C']]\ \rho_0\ \Theta)$

$\text{\$rarrow}_{\text{(G1.1, D1.2)}}\quad \mathbf{L}[[p(\bar{T})]]\ (\lambda x\lambda y\lambda z.(\mathbf{C}[[H_0 :- B_0]]\rho_0\ x\ y\ z\ \text{\$sometimes}\ \mathbf{D}[[C']]\rho_0\ x\ y\ z))\ \Theta)$

$\text{\$rarrow}_{\text{L1.1}}\quad (\lambda x\lambda y\lambda z.((\mathbf{C}[[H_0 :- B_0]]\rho_0\ x\ y\ z)\ \text{\$sometimes}\ (\mathbf{D}[[C']]\rho_0\ x\ y\ z))\ p\ \bar{T})\ \text{\$circle}\ \Theta$

$\text{\$rarrow}\quad \lambda z.((\mathbf{C}[[H_0 :- B_0]]\rho_0\ p\ \bar{T}\ z)\ \text{\$sometimes}\ \mathbf{D}[[C']]\rho_0\ p\ \bar{T}\ z))\ \text{\$circle}\ \Theta$

$\text{\$rarrow}_{\text{C1.1}}\quad \lambda z.((\lambda x.x{\downarrow}dom(z))\ \|\ \mathbf{G}[[B_1]]\rho_0\ \theta{\circ}z :: nil)\ \text{\$sometimes}\ (\mathbf{D}[[C']]\rho_0\ p\ \bar{T}\ z)\ \text{\$circle}\ \Theta$, where

$H_1 :- B_1 = rename((H_0 :- B_0), dom(z))$, $H_1 = p(\bar{S})$, and

$\theta = unify(z(\bar{T}), \bar{S}).$ (1)

The right hand side is

$\lambda\sigma.interp(\text{\$langle\$langle}\, p(\bar{T}) :: nil,\ dom(\sigma)\text{\$rangle} :: nil,\ \sigma,\ (H_0 :- B_0) :: C'\text{\$rangle} :: nil,\ P)\ \text{\$circle}\ \Theta$

$\text{\$rarrow}_{\text{I1.4}}\quad \lambda\sigma.interp(\text{\$langle\$langle}\, B_1,\ dom(\sigma)\text{\$rangle} :: \text{\$langle}\, nil,\ dom(\sigma)\text{\$rangle} :: nil,\ \theta{\circ}\sigma,\ P\text{\$rangle} :: St,\ P)\ \text{\$circle}\ \Theta$, where

$H_1 :- B_1 = rename((H_0 :- B_0, dom(\sigma)), H_1 = p(\bar{S}),$

$\theta = unify(\sigma(\bar{T}), \bar{S}),$

$St = \text{\$langle\$langle}\, p(\bar{T}) :: nil,\ dom(\sigma)\text{\$rangle} :: nil,\ \sigma,\ C'\text{\$rangle} :: nil.$

$\text{\$rarrow}_{\text{Lemmas 3.1, 3.2}}\quad \lambda\sigma.[((\lambda\psi.interp(\text{\$langle\$langle}\, nil,\ dom(\psi)\text{\$rangle} :: nil,\ \psi,\ P\text{\$rangle} :: nil,\ P)\ \text{\$circle}$

$interp(\text{\$langle\$langle}\, B_1,\ dom(\sigma)\text{\$rangle} :: nil,\ \theta{\circ}\sigma,\ P\text{\$rangle} :: nil,\ P))\ \text{\$sometimes}$

$interp(\text{\$langle\$langle}\, p(\bar{T}) :: nil,\ dom(\sigma)\text{\$rangle} :: nil,\ \sigma,\ C'\text{\$rangle} :: nil,\ P)]\ \text{\$circle}\ \Theta$

$\text{\$rarrow}_{\text{I1.6}}\ \lambda\sigma.[(\lambda\psi.\psi{\downarrow}dom(\sigma)\ \|\ interp(\text{\$langle\$langle}\, B_1,\ dom(\sigma)\text{\$rangle} :: nil,\ \theta{\circ}\sigma,\ P\text{\$rangle} :: nil,\ P))\ \text{\$sometimes}$

$interp(\text{\$langle\$langle}\, p(\bar{T}) :: nil,\ dom(\sigma)\text{\$rangle} :: nil,\ \sigma,\ C'\text{\$rangle} :: nil,\ P)]\ \text{\$circle}\ \Theta$

$\text{\$rarrow}\ \lambda\sigma.[(\lambda\psi.\psi{\downarrow}dom(\sigma)\ \|$

$(\lambda\phi.interp(\text{\$langle\$langle}\, B_1,\ dom(\phi)\text{\$rangle} :: nil,\ \phi,\ P\text{\$rangle} :: nil,\ P)\ \text{\$circle}\ \theta{\circ}\sigma :: nil))\ \text{\$sometimes}$

$(\lambda\phi.interp(\text{\$langle\$langle}\, p(\bar{T}) :: nil,\ dom(\phi)\text{\$rangle} :: nil,\ \phi,\ C'\text{\$rangle} :: nil,\ P)\ \text{\$circle}\ \sigma :: nil))]\ \text{\$circle}\ \Theta$ (2)

If $B_1 = nil$, then this yields

$$\mathit{\$rarrow}_{\mathrm{I1.6}} \quad \lambda\sigma.[(\lambda\psi.\psi{\downarrow}\mathit{dom}(\sigma) \parallel (\sigma{\downarrow}\mathit{dom}(\sigma) :: \mathit{nil}))\ \mathit{\$sometimes}$$
$$(\lambda\phi.\mathit{interp}(\mathit{\$langle\$langle}\, p(\bar{T}) :: \mathit{nil}, \mathit{dom}(\phi)\mathit{\$rangle}\,, \phi, C'\mathit{\$rangle}\ :: \mathit{nil}, P)\ \mathit{\$circle}\ \sigma :: \mathit{nil})]\ \mathit{\$circle}\ \Theta$$
$$\mathit{\$rarrow} \quad \lambda\sigma.[(\lambda\psi.\psi{\downarrow}\mathit{dom}(\sigma) \parallel (\theta_\circ\sigma :: \mathit{nil}))\ \mathit{\$sometimes}$$
$$(\lambda\phi.\mathit{interp}(\mathit{\$langle\$langle}\, p(\bar{T}) :: \mathit{nil}, \mathit{dom}(\phi), C'\mathit{\$rangle}\ :: \mathit{nil}, P)\ \mathit{\$circle}\ \sigma :: \mathit{nil})]\ \mathit{\$circle}\ \Theta$$

Now we have

$$\lambda x.x{\downarrow}\mathit{dom}(y) \parallel \mathbf{G}[[B_1]]\rho_0\ (\theta_\circ\sigma :: \mathit{nil})$$
$$\mathit{\$rarrow}\ \ \lambda x.x{\downarrow}\mathit{dom}(y) \parallel \mathbf{G}[[\mathit{nil}_1]]\rho_0\ (\theta_\circ\sigma :: \mathit{nil})$$
$$\mathit{\$rarrow}\ \ \lambda x.x{\downarrow}\mathit{dom}(y) \parallel (\theta_\circ\sigma :: \mathit{nil})$$

and from the induction hypothesis,

$$\mathit{interp}(\mathit{\$langle\$langle}\, p(\bar{T}) :: \mathit{nil}, \mathit{dom}(\phi)\mathit{\$rangle}\ :: \mathit{nil}, \phi, C'\mathit{\$rangle}\ :: \mathit{nil}, P)$$
$$\mathit{\$rarrow}\ \mathrm{BG}[[\mathit{nil}]]\rho_0(\mathbf{L}[[p(\bar{T})]]\ \mathbf{D}[[C']]\rho_0\ \Theta)$$
$$\mathit{\$rarrow}_{\mathrm{G1.1}} \quad \mathbf{L}[[p(\bar{T})]]\ \mathbf{D}[[C']]\rho_0\ \Theta$$
$$\mathit{\$rarrow}_{\mathrm{L1.1}} \quad (\mathbf{D}[[C']]\rho_0\ p\ \bar{T})\ \mathit{\$circle}\ \Theta$$

whence (1) $\equiv$ (2), and the theorem holds.

When $B_1 \neq \mathit{nil}$, assume $B_1 = R :: B_2$. From the induction hypothesis, (2) reduces to

$$\lambda\sigma.[(\lambda\psi.\psi{\downarrow}\mathit{dom}(\sigma) \parallel \mathbf{G}[[B_2]]\ \rho_0\ \mathbf{L}[[R]]\ \mathbf{D}[[P]]\rho_0\ \theta_\circ\sigma :: \mathit{nil})\ \mathit{\$sometimes}$$
$$(\mathbf{G}[[\mathit{nil}]]\rho_0\ \mathbf{L}[[p(\bar{T})]]\ \mathbf{D}[[C']]\rho_0\ \sigma :: \mathit{nil})]\ \mathit{\$circle}\ \Theta$$

where $\rho_0 = \mathbf{D}[[P]]\rho_0$ (being the fixpoint). This can therefore be rewritten, using G1.1, L1.1, as

$$\lambda\sigma.[(\lambda\psi.\psi{\downarrow}\mathit{dom}(\sigma) \parallel \mathbf{G}[[R :: B_2]]\rho_0\ \sigma :: \mathit{nil})\ \mathit{\$sometimes}\ \ (\mathbf{D}[[C']]\rho_0\ p\ \bar{T}\ \sigma)]\ \mathit{\$circle}\ \Theta.$$

The theorem is therefore seen to hold.

Case 3: Here, unification of the literal with the head of the first clause fails. The left hand side is

$$\mathbf{G}[[G]]\ \rho_0\ (\mathbf{L}[[p(\bar{T})]]\ \mathbf{D}[[C]]\ \rho_0\ \Theta)$$
$$\mathit{\$rarrow}_{(\mathrm{G1.1,\ D1.2})} \quad \lambda z.[(\mathbf{C}[[H_0 :\!-\, B_0]]\rho_0\ p\ \bar{T}\ z)\ \mathit{\$sometimes}\ \ (\mathbf{D}[[C']]\rho_0\ p\ \bar{T}\ z)]\ \mathit{\$circle}\ \Theta$$
$$\mathit{\$rarrow}_{\mathrm{C1.1}} \quad \lambda z.[\mathit{nil}\ \mathit{\$sometimes}\ \ (\mathbf{D}[[C']]\rho_0\ p\ \bar{T}\ z)]\ \mathit{\$circle}\ \Theta.$$
$$\mathit{\$rarrow} \quad \lambda z.[\mathbf{D}[[C']]\rho_0\ p\ \bar{T}\ z]\ \mathit{\$circle}\ \Theta.$$

The right hand side is

$$\lambda\sigma.\mathit{interp}(\mathit{\$langle\$langle}\, p(\bar{T}) :: \mathit{nil}, \mathit{dom}(\sigma)\mathit{\$rangle}\ :: \mathit{nil}, \sigma, (H_0 :\!-\, B_0) :: C'\mathit{\$rangle}\ :: \mathit{nil}, P)\ \mathit{\$circle}\ \Theta$$
$$\mathit{\$rarrow}_{\mathrm{I1.5}} \quad \lambda\sigma.\mathit{interp}(\mathit{\$langle\$langle}\, \mathrm{p}(\bar{T}) :: \mathit{nil}, \mathit{dom}(\sigma)\mathit{\$rangle}\ :: \mathit{nil}, \sigma, C'\mathit{\$rangle}\ :: \mathit{nil}, P)\ \mathit{\$circle}\ \Theta$$

which reduces, from the inductive hypothesis, to

$$\mathbf{G}[[\mathit{nil}]]\rho_0\ (\mathbf{L}[[p(\bar{T})\ \mathbf{D}[[C']]\rho_0\ \Theta)$$
$$\mathit{\$rarrow}_{\mathrm{G1.1,\ C1.1)}} \quad \lambda z.[\mathbf{D}[[C']]\rho_0\ p\ \bar{T}\ z]\ \mathit{\$circle}\ \Theta.$$

Case 4: The left hand side is

$$\mathbf{G}[[L :: G_1]]\rho_0\ (\mathbf{L}[[p(\bar{T})]]\ \mathbf{D}[[nil]]\rho_0\ \Theta)$$
$$\textit{\$rarrow}_{\text{(G1.2, D1.1, L1.1)}}\quad \mathbf{G}[[G_1]]\rho_0\ (\mathbf{L}[[L]]\rho_0\ nil)$$
$$\textit{\$rarrow}_{\text{Lemma 3.5}}\qquad nil.$$

The right hand side is

$$\lambda\sigma.interp(\textit{\$langle\$langle}\, p(\bar{T}) :: L :: G_1,\ dom(\sigma)\textit{\$rangle} :: nil,\ \sigma,\ nil\textit{\$rangle} :: nil,\ P)\ \textit{\$circle}\ \Theta$$
$$\textit{\$rarrow}_{\text{I1.3}}\quad \lambda\sigma.interp(nil, P)\ \textit{\$circle}\ \Theta$$
$$\textit{\$rarrow}_{\text{I1.1}}\quad nil.$$

Case 5: The left hand side is

$$\mathbf{G}[[L :: G_1]]\rho_0\ (\mathbf{L}[[p(\bar{T})]]\ \mathbf{D}[[(H_0 :\!-\ B_0) :: C']]\rho_0\ \Theta)$$
$$\textit{\$rarrow}_{\text{(G1.2, D1.2, C1.1)}}\quad \mathbf{G}[[G_1]]\rho_0\ (\mathbf{L}[[L]]\rho_0\ (\lambda z.[(\lambda x.x{\downarrow}dom(z)\ \|$$
$$\mathbf{G}[[B_1]]\rho_0\ \theta\circ z :: nil)\ \textit{\$sometimes}\ (\mathbf{D}[[C']]\rho_0\ p\ \bar{T}\ z)]\ \textit{\$circle}\ \Theta)),\ \text{where}$$
$$H_1 :\!-\ B_1 = rename((H_0 :\!-\ B_0), dom(z)),\ H_1 = p(\bar{S}),\ \text{and}$$
$$\theta = unify(z(\bar{T}), \bar{S}).$$

The right hand side is

$$\lambda\sigma.interp(\textit{\$langle\$langle}\, p(\bar{T}) :: L :: G_1,\ dom(\sigma)\textit{\$rangle} :: nil,\ \sigma,\ (H_0 :\!-\ B_0) :: C'\textit{\$rangle} :: nil,\ P)\ \textit{\$circle}\ \Theta$$
$$\textit{\$rarrow}_{\text{I1.4}}\quad \lambda\sigma.interp(\textit{\$langle\$langle}\, B_1,\ dom(\sigma)\textit{\$rangle} :: \textit{\$langle}\, L :: G_1,\ dom(\sigma)\textit{\$rangle} :: nil,\ \theta\circ\sigma,\ P\textit{\$rangle} :: St,\ P),\ \text{where}$$
$$H_1 :\!-\ B_1 = rename((H_0 :\!-\ B_0), dom(z)),\ H_1 = p(\bar{S}),$$
$$\theta = unify(\sigma(\bar{T}), \bar{S}),\ \text{and}$$
$$St = \textit{\$langle\$langle}\, p(\bar{T}) :: nil,\ dom(\sigma)\textit{\$rangle} :: nil,\ \sigma,\ C'\textit{\$rangle} :: nil.$$

That the theorem holds now follows from Lemmas 3.1, 3.2, 3.3 and the inductive hypothesis.

Case 6: Here, unification of the leftmost literal of the goal with the head of the first clause fails. The left hand side is

$$\mathbf{G}[[L :: G_1]]\rho_0\ (\mathbf{L}[[p(\bar{T})]]\ \mathbf{D}[[(H_0 :\!-\ B_0) :: C']]\rho_0\ \Theta)$$
$$\textit{\$rarrow}_{\text{(G1.2, D1.2, L1.1)}}\quad \mathbf{G}[[G]]\rho_0\ (\lambda y.[\mathbf{D}[[C']]p\ \bar{T}\ y]\ \textit{\$circle}\ \Theta)).$$

The right hand side is

$$\lambda\sigma.interp(\textit{\$langle\$langle}\, p(\bar{T}) :: L :: G_1,\ dom(\sigma)\textit{\$rangle} :: nil,\ \sigma,\ (H_0 :\!-\ B_0) :: C'\textit{\$rangle} :: nil,\ P)\ \textit{\$circle}\ \Theta$$
$$\textit{\$rarrow}_{\text{I1.4}}\quad \lambda\sigma.interp(\textit{\$langle\$langle}\, p(\bar{T}) :: L :: G_1,\ dom(\sigma)\textit{\$rangle} :: nil,\ \sigma,\ C'\textit{\$rangle} :: nil,\ P)\ \textit{\$circle}\ \Theta$$

and the theorem now follows from Lemma 3.3 and Case 3 above. *\$always*

Corollary (Equivalence of Denotational and Operational Semantics): For any goal G and program P, and substitution sequence Θ,

$$\mathbf{G}[[G]]\rho_0\,\Theta \equiv \lambda\sigma.interp(\$langle\$langle\, G, dom(\sigma)\$rangle :: nil, \sigma, P\$rangle :: nil, P)\ \$circle\ \Theta.$$

Proof: The case where $G = nil$ follows directly from the definitions of **G**[[ ]] and *interp*. The case where $G \neq nil$ follows from Theorem 3.1, with $C = P$. *$always*

Appendix 2: Proofs of Equivalence for Prolog with Cut

Lemma 4.4: For any framelist F, substitution sequence $\Phi_1\ \mathit{\$sometimes}\ \Phi_2$, stack component St and program P,

$$\mathit{tran}(\mathit{\$langle}\, F, \Phi_1\ \mathit{\$sometimes}\ \Phi_2, P\mathit{\$rangle} :: St, P) = \mathit{tran}(\mathit{\$langle}\, F, \Phi_1, P\mathit{\$rangle} :: \mathit{\$langle}\, F, \Phi_2, P\mathit{\$rangle} :: St, P).$$

Proof: By structural induction on F.

The base case, with $F = \mathit{nil}$, follows directly from the definition of *tran*. For the induction step, consider $F = \mathit{\$langle}\, G, V_P, D\mathit{\$rangle} :: \mathit{FRest}$. The left hand side is

$$\begin{aligned}
&\text{let } \mathit{\$langle}\, \Psi, \mathit{cf}\mathit{\$rangle} = \hat{\mathbf{G}}[[G]]\rho_0\, \rho_0\, \mathit{\$langle}\, \Phi_1\ \mathit{\$sometimes}\ \Phi_2, \mathbf{f}\mathit{\$rangle} \text{ in} \\
&\qquad \text{if } \Psi = \mathit{nil} \text{ then } \mathit{tran}(\text{if } \mathit{cf} \text{ then } D \text{ else } St, P) && (1)\\
&\qquad \text{else } \mathit{tran}(\mathit{FRest}, (\lambda x.x{\downarrow}V_P \parallel \Psi), P\mathit{\$rangle} :: (\text{if } \mathit{cf} \text{ then } D \text{ else } St), P). && (2)
\end{aligned}$$

Since $\Phi_1 \le \Phi_1\ \mathit{\$sometimes}\ \Phi_2$, if $\mathit{\$langle}\, \Psi_1, \mathit{cf}_1\mathit{\$rangle} = \hat{\mathbf{G}}[[G]]\rho_0\, \rho_0\, \mathit{\$langle}\, \Phi_1, \mathbf{f}\mathit{\$rangle}$ then, from Lemma 4.1, $\Psi_1 \le \Psi$. Let $\Psi = \Psi_1\ \mathit{\$sometimes}\ \Psi_2$. Then (2) reduces to

$$\begin{aligned}
&\mathit{tran}(\mathit{\$langle}\, \mathit{FRest}, (\lambda x.x{\downarrow}V_P) \parallel \Psi_1)\ \mathit{\$sometimes}\ (\lambda x.x{\downarrow}V_P \parallel \Psi_2), P\mathit{\$rangle} :: (\text{if } \mathit{cf} \text{ then } D \text{ else } St), P) \\
&\mathit{\$arrow}_{\text{Inductive Hypothesis}}\ \mathit{tran}(\mathit{\$langle}\, \mathit{FRest}, (\lambda x.x{\downarrow}V_P) \parallel \Psi_1, P\mathit{\$rangle} :: \\
&\qquad\qquad \mathit{\$langle}\, \mathit{FRest}, (\lambda x.x{\downarrow}V_P) \parallel \Psi_2, P\mathit{\$rangle} :: (\text{if } \mathit{cf} \text{ then } D \text{ else } St), P) && (3)
\end{aligned}$$

The right hand side is

$$\begin{aligned}
&\text{let } \mathit{\$langle}\, \Psi_1, \mathit{cf}_1\mathit{\$rangle} = \hat{\mathbf{G}}[[G]]\rho_0\, \rho_0\, \mathit{\$langle}\, \Phi_1, \mathbf{f}\mathit{\$rangle} \text{ in} \\
&\quad \text{if } \Psi_1 = \mathit{nil} \text{ then } \mathit{tran}(\text{if } \mathit{cf}_1 \text{ then } D \text{ else } \mathit{\$langle}\, F, \Phi_2, D\mathit{\$rangle} :: St, P) && (A)\\
&\quad \text{else } \mathit{tran}(\mathit{\$langle}\, \mathit{FRest}, (\lambda x.x{\downarrow}V_P) \parallel \Psi_1, P\mathit{\$rangle} :: (\text{if } \mathit{cf}_1 \text{ then } D \text{ else } \mathit{\$langle}\, F, \Phi_2, D\mathit{\$rangle} :: St, P) && (B)
\end{aligned}$$

We will now systematically show that cases (A) and (B) yield the same results as (1) and (3).

Case A: $\Psi_1 = \mathit{nil}$:

(i) If cf_1 is true then cf must also be true, whence (A) $\equiv$ (1).

If cf_1 is false then (A) reduces to

$$\begin{aligned}
&\mathit{tran}(\mathit{\$langle}\mathit{\$langle}\, G, V_P, D\mathit{\$rangle} :: \mathit{FRest}, \Phi_2, P\mathit{\$rangle} :: St, P) \\
&\mathit{\$arrow}_{\text{T2.4}} \quad \text{let } \mathit{\$langle}\, \Psi', \mathit{cf}'\mathit{\$rangle} = \hat{\mathbf{G}}[[G]]\rho_0\, \rho_0\, \mathit{\$langle}\, \Phi_2, \mathbf{f}\mathit{\$rangle} \text{ in} \\
&\qquad \text{if } \Psi' = \mathit{nil} \text{ then } \mathit{tran}(\text{if } \mathit{cf}' \text{ then } D \text{ else } St, P) && (A1)\\
&\qquad \text{else } \mathit{tran}(\mathit{\$langle}\, \mathit{FRest}, (\lambda x.x{\downarrow}V_P) \parallel \Psi', P\mathit{\$rangle} :: (\text{if } \mathit{cf}' \text{ then } D \text{ else } St), P). && (A2)
\end{aligned}$$

If Ψ' is *nil*, there are two possibilities: if cf' is true then cf is true, and (A1) reduces to $\mathit{tran}(D, P)$, which is the same as (1). On the other hand, if cf' is false, then cf is false, and (A1) reduces to $\mathit{tran}(St, P)$, which is again the same as (1).

If $\Psi' \neq nil$, then from Lemma 4.2, cf_1 is false implies $\Psi' = \Psi_2$, and (A2) = (3) with $\Psi_1 = nil$.

Case B: $\Psi_1 \neq nil$:

If cf_1 is true then cf is also true. (B) reduces to $tran(\text{\$langle}\, FRest,\ (\lambda x.x{\downarrow}V_P) \parallel \Psi_1,\ P\text{\$rangle} :: D,\ P)$, while (2) reduces to $tran(\text{\$langle}\, FRest,\ ((\lambda x.x{\downarrow}V_P) \parallel \Psi_1)\ \text{\$sometimes}\ ((\lambda x.x{\downarrow}V_P \parallel \Psi_2),\ P\text{\$rangle} :: D,\ P)$. From Lemma 4.3, $\Psi_2 = nil$, and (B) = (2).

If cf_1 is false, then (B) reduces to $tran(\text{\$langle}\, FRest,\ (\lambda x.x{\downarrow}V_P) \parallel \Psi_1,\ P\text{\$rangle} :: \text{\$langle}\, F,\ \Phi_2,\ P\text{\$rangle} :: St,\ P)$, while (3) reduces to $tran(\text{\$langle}\, FRest,\ (\lambda x.x{\downarrow}V_P) \parallel \Psi_1,\ P\text{\$rangle} :: \text{\$langle}\, FRest,\ (\lambda x.x{\downarrow}V_P) \parallel \Psi_2,\ P\text{\$rangle} :: St,\ P)$. From Lemma 4.2, observe that if cf_1 is false then $\Psi_2 = \Psi'$, whence (B) = (3). *\$always*

Theorem 4.1: For any framelist F, substitution θ, clause list C, stack component St and program P,

$$interp(\text{\$langle}\, F,\ \theta,\ C\text{\$rangle} :: St,\ P) \equiv tran(\text{\$langle}\, F,\ \theta :: nil,\ C\text{\$rangle} :: St,\ P).$$

Proof: By fixpoint induction on *interp* and *tran*. The base case, where both *tran* and *interp* have empty stacks, follows from their definitions. The inductive step has five cases:

Case 1: $F = nil$.

Case 2: $F \neq nil$, $C = nil$.

Case 3: $F = \text{\$langle}\, G,\ V_P,\ D\text{\$rangle} :: FRest$, $C \neq nil$, $G = $ ‘!’ $:: G'$.

Case 4: $F = \text{\$langle}\, G,\ V_P,\ D\text{\$rangle} :: FRest$, $C \neq nil$, $G = p(\overline{T}) :: G'$, unification of $p(\overline{T})$ with the first clause of C fails.

Case 5: $F = \text{\$langle}\, G,\ V_P,\ D\text{\$rangle} :: FRest$, $C \neq nil$, $G = p(\overline{T}) :: G'$, unification of $p(\overline{T})$ with the first clause of C succeeds.

The proof proceeds as follows:

Case 1: $F = nil$: the left hand side is $interp(\text{\$langle}\, nil,\ \theta,\ C\text{\$rangle} :: St,\ P)$

$\text{\$rarrow}_{\mathrm{I2.2}}\quad \theta :: interp(St,\ P)$,

while the right hand side is $tran(\text{\$langle}\, nil,\ \theta :: nil,\ C\text{\$rangle} :: St,\ P)$

$\text{\$rarrow}_{\mathrm{T2.2}}\quad (\theta :: nil)\ \text{\$sometimes}\ tran(St,\ P)$
$\text{\$rarrow}\quad \theta :: tran(St,\ P)$

and from the induction hypothesis, the theorem holds.

Case 2: The left hand side is $interp(\text{\$langle}\, F_0 :: FRest,\ \theta,\ nil\text{\$rangle} :: St,\ P)$

$\text{\$rarrow}_{\mathrm{I2.3}}\quad interp(St,\ P)$.

The right hand side is $tran(\$langle\, F_0 :: FRest, \theta :: nil, nil\$rangle :: St, P)$

$\$rarrow_{\mathrm{T2.3}} \quad tran(St, P)$.

From the induction hypothesis, the theorem holds.

Case 3: The left hand side is

$interp(\$langle\$langle\, \text{'!'} :: G, V_P, D\$rangle :: FRest, \theta, C\$rangle :: St, P)$

$\$rarrow_{\mathrm{I2.4}} \quad interp(\$langle\$langle\, G, V_P, D\$rangle :: FRest, \theta, P\$rangle :: D, P)$.

On right hand side, we have

$\$langle\, \Phi, cflag\$rangle = \hat{\mathbf{G}}[[! :: nil]]\rho_0\ (\mathbf{D}[[C]]\$langle\, \rho_0, ncont\$rangle)\ \$langle\, \theta :: nil, \mathbf{f}\$rangle$

$\$rarrow_{\mathrm{(G2.2,\ L2.2)}} \quad \$langle\, \Phi, cflag\$rangle = \$langle\, \theta :: nil, \mathbf{t}\$rangle$

From this, the right hand side reduces to $tran(\$langle\$langle\, G, V_P, D\$rangle :: FRest, \theta :: nil, P\$rangle :: D, P)$, and from the induction hypothesis, we are done.

Case 4: $F = \$langle\, p(\overline{T}) :: G, V_P, D\$rangle :: FRest$, $C = (H_0 :- B_0) :: C'$, $H_1 :- B_1 = rename((H_0 :- B_0), dom(\theta))$, and $unify(\theta(p(\overline{T})), H_1) = fail$.

The left hand side is

$interp(\$langle\$langle\, p(\overline{T}) :: G, V_P, D\$rangle :: FRest, \theta, (H_0 :- B_0) :: C'\$rangle :: St, P)$

$\$rarrow_{\mathrm{I2.6}} \quad interp(\$langle\$langle\, p(\overline{T}) :: G, V_P, D\$rangle :: FRest, \theta, C'\$rangle :: St, P)$

$\$rarrow_{\mathrm{Induction\ Hypothesis}}\ tran(\$langle\$langle\, p(\overline{T}) :: G, V_P, D\$rangle :: FRest, \theta :: nil, C'\$rangle :: St, P)$.

$\$rarrow_{\mathrm{T2.4}}$ let $\$langle\, \Phi, cflag\$rangle = \hat{\mathbf{G}}[[p(\overline{T}) :: nil]]\rho_0\ (\mathbf{D}[[C']]\$langle\, \rho_0, ncont\$rangle)\ \$langle\, \theta :: nil, \mathbf{f}\$rangle$ in

if $\Phi = nil$ then $tran($if $cflag$ then D else $St, P)$;

if $\Phi \neq nil$ then

$tran(\$langle\$langle\, G, V_P, D\$rangle :: FRest, \Phi, P\$rangle :: ($if $cflag$ then D else $St), P)$ (1)

Now $\$langle\, \Phi, cflag\$rangle = \hat{\mathbf{G}}[[p(\overline{T}) :: nil]]\rho_0\ (\mathbf{D}[[C']]\$langle\, \rho_0, ncont\$rangle)\ \$langle\, \theta :: nil, \mathbf{f}\$rangle$

$\$rarrow_{\mathrm{(L2.2,\ G2.1)}} \quad \$langle\, ((\mathbf{D}[[C']]\$langle\, \rho_0, ncont\$rangle)\ p\ \overline{T}\ \theta), \mathbf{f}\$rangle$

and since $cflag = \mathbf{f}$, (1) reduces to

if $\Phi = nil$ then $tran(St, P)$ else $tran(\$langle\$langle\, G, V_P, D\$rangle :: FRest, \Phi, P\$rangle :: St, P)$.

The right hand side is

$tran(\$langle\$langle\, p(\overline{T}) :: G, V_P, D\$rangle :: FRest, \theta :: nil, (H_0 :- B_0) :: C'\$rangle :: St, P)$

$\$rarrow_{\mathrm{T2.4}}$ let $\$langle\, \Phi, cflag\$rangle = \hat{\mathbf{G}}[[p(\overline{T}) :: nil]]\rho_0\ (\mathbf{D}[[(H_0 :- B_0) :: C']]\$langle\, \rho_0, ncont\$rangle)\ \$langle\, \theta :: nil, \mathbf{f}\$rangle$

in if $\Phi = nil$ then $tran($if $cflag$ then D else $St, P)$;

if $\Phi \neq nil$ then

$$\mathit{tran}(\text{\$langle\$langle}\, G, V_P, D\text{\$rangle} :: \mathit{FRest}, \Phi, P\text{\$rangle} :: (\text{if } \mathit{cflag} \text{ then } D \text{ else } \mathit{St}), P)$$

Since unification of $p(\bar{T})$ with the head of the first clause fails, it can effectively be discarded:

$$\hat{\mathbf{G}}[[p(\bar{T}) :: \mathit{nil}]]\rho_0\, (\mathbf{D}[[(H_0 :\!-\, B_0) :: C']]\text{\$langle}\, \rho_0, \mathit{ncont}\text{\$rangle}\,)\, \text{\$langle}\, \theta :: \mathit{nil}, \mathbf{f}\text{\$rangle}$$

$$\text{\$rarrow}_{\mathrm{D2.2}} \quad \hat{\mathbf{G}}[[p(\bar{T}) :: \mathit{nil}]]\rho_0\, (\mathbf{C}[[H_0 :\!-\, B_0]]\text{\$langle}\, \rho_0, \mathbf{D}[[C']]\text{\$langle}\, \rho_0, \mathit{ncont}\text{\$rangle\$rangle}\,)\, \text{\$langle}\, \theta :: \mathit{nil}, \mathbf{f}\text{\$rangle}$$

$$\text{\$rarrow}_{(\mathrm{L2.2,\ C2.1})} \quad \text{\$langle}\, ((\mathbf{D}[[C']]\text{\$langle}\, \rho_0, \mathit{ncont}\text{\$rangle}\,)\ p\ \bar{T}\ \theta), \mathbf{f}\text{\$rangle}\,.$$

Thus, from the above, the left and right hand sides are equal, so we are done.

Case 5: $F = \text{\$langle}\, p(\bar{T}) :: G, V_P, D\text{\$rangle} :: \mathit{FRest}$, $C = (H_0 :\!-\, B_0) :: C'$,

$H_1 :\!-\, B_1 = \mathit{rename}((H_0 :\!-\, B_0), \mathit{dom}(\theta))$, $\phi = \mathit{unify}(\theta(p(\bar{T})), H_1) \neq \mathit{fail}$.

On the right hand side, we have

$$\text{\$langle}\, \Phi, \mathit{cflag}\text{\$rangle} = \hat{\mathbf{G}}[[p(\bar{T}) :: \mathit{nil}]]\rho_0\, (\mathbf{D}[[C]]\text{\$langle}\, \rho_0, \mathit{ncont}\text{\$rangle}\,)\, \text{\$langle}\, \theta :: \mathit{nil}, \mathbf{f}\text{\$rangle}$$

$$\text{\$rarrow}_{(\mathrm{D2.2,\ C2.1,\ L2.2})} \quad \text{\$langle if } \mathit{cflag} \text{ then } \lambda x.x{\downarrow}\mathit{dom}(\theta) \parallel \Psi \text{ else } (\lambda x.x{\downarrow}\mathit{dom}(\theta) \parallel \Psi)\ \text{\$sometimes}\ \Theta, \mathbf{f}\text{\$rangle}$$

where $\text{\$langle}\, \Psi, \mathit{cflag}_0\text{\$rangle} = \mathbf{G}[[B_1]]\rho_0\, \text{\$langle}\, \phi\circ\theta :: \mathit{nil}, \mathbf{f}\text{\$rangle}$, and

$$\Theta = (\mathbf{D}[[C']]\text{\$langle}\, \rho_0, \mathit{ncont}\text{\$rangle}\,)\ p\ \bar{T}\ \theta \qquad (2)$$

Since $\mathit{cflag} = \mathbf{f}$, the right hand side therefore reduces to

$$\text{if } \Phi = \mathit{nil} \text{ then } \mathit{tran}(\mathit{St}, P) \qquad (3)$$

$$\text{else } \mathit{tran}(\text{\$langle\$langle}\, G, V_P, D\text{\$rangle} :: \mathit{FRest}, \Phi, P\text{\$rangle} :: \mathit{St}, P). \qquad (4)$$

The left hand side is

$$\mathit{interp}(\text{\$langle\$langle}\, p(\bar{T}) :: G, V_P, D\text{\$rangle} :: \mathit{FRest}, \theta, C\text{\$rangle} :: \mathit{St}, P)$$

$$\text{\$rarrow}_{\mathrm{I2.5}} \quad \mathit{interp}(\text{\$langle\$langle}\, B_1, \mathit{dom}(\theta), \mathit{St}\text{\$rangle} :: \text{\$langle}\, G, V_P, D\text{\$rangle} :: \mathit{FRest}, \phi\circ\theta, P\text{\$rangle} :: \mathit{St}', P)$$

where $\mathit{St}' = \text{\$langle\$langle}\, p(\bar{T}) :: G, V_P, D\text{\$rangle} :: \mathit{FRest}, \theta, C'\text{\$rangle} :: \mathit{St}$.

$$\text{\$rarrow}_{\text{Inductive Hypothesis}} \text{ let } \text{\$langle}\, \Psi, \mathit{cflag}_0\text{\$rangle} = \mathbf{G}[[B_1]]\rho_0\, \text{\$langle}\, \phi\circ\theta :: \mathit{nil}, \mathbf{f}\text{\$rangle}$$

in

$$\text{if } \Psi = \mathit{nil}\ \text{\$and}\ \mathit{cflag}_0 \text{ then } \mathit{tran}(\mathit{St}, P) \qquad \text{(A)}$$

$$\text{else if } \Psi = \mathit{nil}\ \text{\$and}\ \neg\mathit{cflag}_0 \text{ then } \mathit{tran}(\mathit{St}', P) \qquad \text{(B)}$$

$$\text{else if } \Psi \neq \mathit{nil}\ \text{\$and}\ \mathit{cflag}_0 \text{ then}$$

$$\mathit{tran}(\text{\$langle\$langle}\, \mathit{nil}, \mathit{dom}(\theta), \mathit{St}\text{\$rangle} :: \text{\$langle}\, G, V_P, D\text{\$rangle} :: \mathit{FRest}, \Psi, P\text{\$rangle} :: \mathit{St}, P) \qquad \text{(C)}$$

$$\text{else if } \Psi \neq \mathit{nil}\ \text{\$and}\ \neg\mathit{cflag} \text{ then}$$

$$\mathit{tran}(\text{\$langle\$langle}\, \mathit{nil}, \mathit{dom}(\theta), \mathit{St}\text{\$rangle} :: \text{\$langle}\, G, V_P, D\text{\$rangle} :: \mathit{FRest}, \Psi, P\text{\$rangle} :: \mathit{St}', P) \qquad \text{(D)}$$

We therefore have four subcases to consider:

(1) $\Psi = \mathit{nil}\ \text{\$and}\ \mathit{cflag}_0$: Then, $\Phi = \mathit{nil}$, and (3) = (A). The theorem holds.

(2) $\Psi = nil\ \mathit{\$and}\ \neg cflag_0$: (B) is, from the definition of St',

$$tran(\mathit{\$langle\$langle}\, p(\bar{T}) :: G, V_p, D\mathit{\$rangle} :: FRest, \theta :: nil, C'\mathit{\$rangle} :: St, P.$$

Let $\mathit{\$langle}\, \Xi, cflag_1 \mathit{\$rangle} = \hat{\mathbf{G}}[[p(\bar{T}) :: nil]]\rho_0\, (\mathbf{D}[[C']]\mathit{\$langle}\, \rho_0, ncont\mathit{\$rangle})\ \mathit{\$langle}\, \theta :: nil, \mathbf{f}\mathit{\$rangle}$
$= \mathit{\$langle}\, ((\mathbf{D}[[C']]\mathit{\$langle}\, \rho_0, ncont\mathit{\$rangle})\ p\ \bar{T}\ \theta), \mathbf{f}\mathit{\$rangle}$.

Observe that $\Xi = \Theta$. We therefore have, if $\Theta = nil$, that

$$[\Psi = nil\ \mathit{\$and}\ \neg cflag_0\ \mathit{\$and}\ \Theta = nil] \text{ implies } \Phi = nil$$

and (B) reduces to $tran(St, P)$, which is the same as (3). On the other hand, if $\Theta \neq nil$, then

$$[\Psi = nil\ \mathit{\$and}\ \neg cflag_0\ \mathit{\$and}\ \Theta \neq nil] \text{ implies } \Phi \neq nil$$

and (B) reduces to

$$tran(\mathit{\$langle\$langle}\, G, V_p, D\mathit{\$rangle} :: FRest, \Phi, P\mathit{\$rangle} :: St, P)$$

which is equal to (3). Thus, the theorem holds.

(3) $\Psi \neq nil\ \mathit{\$and}\ cflag_0$: Then, from (T2.5) in the definition of *tran*, (C) reduces to

$$tran(\mathit{\$langle\$langle}\, G, V_p, D\mathit{\$rangle} :: FRest, ((\lambda x.x{\downarrow}dom(\theta)) \parallel \Psi), P\mathit{\$rangle} :: St, P).$$

Observe that if $cflag_0 = \mathbf{t}$ then, from (2), we have $\Phi = (\lambda x.x{\downarrow}dom(\theta)) \parallel \Psi$, so that this becomes

$$tran(\mathit{\$langle\$langle}\, G, V_p, D\mathit{\$rangle} :: FRest, \Phi, P\mathit{\$rangle} :: St, P),$$

which is the same as (4). Thus, the theorem holds.

(4) $\Psi \neq nil\ \mathit{\$and}\ \neg cflag_0$: Then, from (T2.5) in the definition of *tran*, (D) reduces to

$$tran(\mathit{\$langle\$langle}\, G, V_p, D\mathit{\$rangle} :: FRest, ((\lambda x.x{\downarrow}dom(\theta)) \parallel \Psi), P\mathit{\$rangle} :: St', P).$$

That the left and right hand sides are equal now follows from Lemma 4.4. *$always*