Stepwise Development of Operational and Denotational Semantics for PROLOG

Neil D. Jones

Datalogisk Institut
Sigurdsgade 41
DK-2200 Copenhagen, Denmark

Alan Mycroft*

Department of Computer Science
Edinburgh University

ABSTRACT

It is argued that the technique of stepwise refinement applies just as well to semantics as to programming. Accordingly we develop concise and exact characterisations of the semantics of logic programs (with the usual depth first search strategy) starting from resolution as a base. The resulting operational semantics closely resembles traditional implementations, and the denotational version provides the first denotational semantics og PROLOG as used in practice. Confidence in the correctness of the semantics comes from their systematic development from PROLOG's mathematical foundations.

1 INTRODUCTION

The PROLOG language [8] has never benefited from having a proper semantics, but has been used on the assumption that its procedural semantics (i. e. the rules by which answers to a computation are derived) agrees with the interpretation of PROLOG programs as Horn clauses in first-order logic. One can see the incorrectness of this assumption by observing that the standard depth-first procedural interpretation may take more than ω steps to discover that a goal is satisfiable as specified by the logic interpretation, whereas computationally this must be regarded as $\perp$ or the empty set of results.

Apt and van Emden have provided [1] several precise characterisations of the semantics of Horn clauses (the formal equivalent in logic of PROLOG programs), including the SLD-trees used in this paper. They stop short, however of presenting a semantics of PROLOG as it is actually used, as a deterministic programming language.

In this paper we specify the semantics of PROLOG in two ways: a (direct) Scott-Strachey style denotational semantics and an operational semantics expressed as an SECD-style interpreter which is suitable for computer implementation. The two semantics are developed from Kowalski's SLD-refutation procedure [8,9] by sequences of semantics-preserving transformations. The results represent, to our knowledge, the first computational semantics of PROLOG which are well-founded in the underlying theory. Moreover, both our operational and denotational semantics are extremely concise, and we believe that the derivation from a mathematical rule has contributed to this.

We provide in addition the first formalisation of the *cut* operator (which finds its way into most practical PROLOG implementations for reasons of efficiency and its ability to exert some control on the pure backtracking implementation as given by Kowalski). We propose the denotational version of the semantics as a definition of (depth-first) PROLOG with *cut*, observing that the *cut* operation rits naturally into this form and produces little increase in complexity.

We also use the derivation of the denotational form of PROLOG semantics to discuss what comprises a correct denotational definition and the nature of the difference between direct- and continuation--style semantics.

Related work includes an operational semantics due to Komorowski [7] (expressed in the META-IV metalanguage from the Vienna development method [2]) and the informal development of a PROLOG interpreter by van Emden [6]. Nilsson's META IV semantics [13] for PROLOG uses substitution sequences similar to those of Section 7.

2 OVERVIEW

The traditional use of resolution is to show that a formula is a theorem by asserting axioms and the negation of the formula, and deriving a contradiction. Our goal formula will be written as a conjunction $A_1,\ldots,A_n$ and the axioms as *clauses* of the form $C_1,\ldots,C_p \leftarrow B_1,\ldots,B_m$, where A_i, B_j and C_k are atomic formulas (which we call "atoms" as in [1]). Such a clause is interpreted as $C_1 \vee \ldots \vee C_p \subset B_1 \wedge \ldots \wedge B_m$, with an implicit universal quantification over the variables of the C_i's and B_j's.

The axioms are then extended with the negative (p=0) clause $\leftarrow A_1,\ldots,A_n$ which denies $A_1 \wedge \ldots \wedge A_n$ and resolution is applied with the hope of exhibiting a contradiction in the augmented axiom set. A PROLOG program has the form P;N where P is a finite sequence of definite (p=1) or *Horn* clauses and N is a negative clause (the *goal*). A nondeterministic interpreter is easily defined; its state is the current goal formula and its computation steps consist of replacing the current goal by one of its resolvents repeatedly until $\leftarrow$ is obtained.

---

* Current address: Informationsbehandling
Chalmers Tekniska Högskola
S-41296 Göteborg, Sweden

CH2007-3/84/0000/0281 $01.00 © 1984 IEEE

Any real implementation of semantics must incorporate some form of search process to express this mechanism. To this end Apt and van Emden [1] have defined an *SLD-tree* which organises many resolution sequences into a single data structure, and have shown any SLD-tree to be a complete search space: goal $\leftarrow A_1,\ldots,A_n$ is refutable if and only if some SLD-tree with the goal as root contains a contradictory clause. Moreover, if one SLD-tree for a given goal contains such a contradictory clause then all SLD-trees for that goal do. Both our operational and denotational semantics will be based on this notion of SLD-tree traversal.

A major desire is to give some form of formal specification of the *cut* operator in PROLOG. This, unfortunately, is only well-defined with respect to some particular search strategy (=evaluation order). In this paper we define it with respect to the usual depth-first left-right traversal, although others are certainly possible (see [9, 17]). We have no great philosophical bias in choosing a definition of depth-first PROLOG, but observe that it is the current implementation choice and that *cut* appears not to be well-defined for any other evaluation order.

One further difference between PROLOG and resolution theorem proving, is that we are not merely interested in the refutation of a theorem, but also in the particular values of the free variables of the theorem that cause this refutation. Such answer values are naturally expressed as the composition of the substitutions (unifiers) obtained from the resolution steps in the computation. There may be more than one distinct such substitutions which refute the augmented axiom set (or none) and most PROLOG interpreters will provide them as a sequence as they are encountered during the backtracking process. We will actually *define* the output of a PROLOG program to be this (possibly infinite) sequence of substitutions.

The paper is structured according to the following scheme: Section 3 contains the syntax of our version of PROLOG and various notations and minor definitions, and Section 4 provides an introduction to SLD-resolution and its treatment as a computation step. Section 5 and 7 both develop successive semantics for PROLOG, respectively in operational (SECD-style as in [10]) and denotational formulation. The semantics are developed from a simple specification of SLD-tree traversal into respectively a sophisticated interpreter and a denotational semantics. We take great care in both the operational and denotational cases to make our initial semantics as close as possible to SLD-resolution, and then to derive successive semantics by (we hope) obviously correct transformations. We believe that this stepwise development of closely related semantics obviates the need for complicated (and unreadable) proofs of equivalence of different semantic formulations of the same language. This is an analogous idea to Burstall and Darlington's [4] successive program transformations in which simple transformation steps are proved correct and then composed, safe in the knowledge that correctness of such a composition is inherent.

In a succeeding paper we will show how an efficient compiler can be derived from the operational semantics by use of the compiler generator system described in [5].

3 NOTATION AND AUXILIARY DEFINITIONS

3.1 Mathematical Preliminaries

For any set, α say, we will use the notation α^* to stand for the set of finite sequences of elements of α. It can be defined by

$$\alpha^* = \{nil\} + \alpha \times \alpha^* .$$

where + is the disjoint union operator. Objects in the right summand will be constructed by means of a function which will be written as an infix operator :: (to be considered right-associative). We will follow the popular style and write [] for nil and [a,b,...,z] for a::[b,...,z] where this aids readability.

Another useful function associated with :: is the append function, written as infix @, and defined by

```
@: α* × α* → α*
[] @ a = a
(a::b) @ c = a :: (b@c)
```

3.2 Syntax of PROLOG

In the following we assume the existence of disjoint sets of symbols representing names of predicates (Pred), functors (Functor) and variables (Var).

In the following the notation for example) t:Term will indicate that meta-variables with name t (possibly subscripted) will be used to range over Term. The abstract syntax of PROLOG that we consider is constructed from the following rules:

```
t: Term ::= Var + Functor(Term*)
a: Atom ::= Pred(Term*)
b: Body ::= (Atom + {"!"})*
c: Clause ::= Atom ← Body.
P: Sentence ::= Clause*
   Program  ::= Sentence; ← Body
```

The letter q will be used to range over the *query* in the program P;←q. We follow Apt and van Emden and use the name Atom to refer to *atomic formulae* which are compound structures, which conflicts with the other use of "atom" denoting an indivisible object. "!" will be referred to as the *cut* symbol.

For clarity and to indicate the logical context, $\wedge$ and *true* will be used (consistently) as replacements for :: and [] when constructing the sequence of Atoms which constitute a Body, so $[a_1,a_2]$ will be written $a_1 \wedge a_2 \wedge true$.

3.3 Substitutions, Renamings and Unifiers

The set Subst of substitutions (ranged over by ϕ and θ) is defined to be the function space

```
Subst = Var → Term.
```

The set Rename of renamings is that subset of Subst given by the injective maps in the function space

```
Rename = Var → Var
```

and ranged over by Ψ. We will permit substitutions and renamings to act on Term, Atom, lists of Atom and the like by the standard free extension. We

differ from Apt and van Emden's notation in our use of prefix application for substitutions instead of their postfix application. This enables us to use the standard notation for function composition to compose substitutions, so $(\theta\circ\phi)t = \theta(\phi(t))$ for any term t. The identity substitution will be written Id. As we noted above, sequences of substitutions are the natural form for the result of a PROLOG program.

In order to guarantee that an atom and a clause selected for resolution against that atom have no variables in common, we wish to rename the clause so that this is indeed the case. Our method for performing this is to partition Var into a countable number of disjoint sets $\{Var_0, Var_1, \ldots\}$ which are all isomorphic, with Var_0 containing all the variables which may validly occur in the program source. We then isolate a sequence of bijective renamings Ψ_n: $Var_0 \rightarrow Var_n$ $(n>0)$ which will be used to rename the clauses selected for resolution. Ψ_0 is the identity function.

We assume the existence of a unification function as first exhibited by Robinson [15]. Manna and Waldinger give much insight in their recent work [11] which examines unification very carefully. The unification algorithm gives a function

$$MGU: Atom \times Atom \rightarrow Subst + \{fail\}$$

such that $MGU(a_1,a_2)$ is the most general unifier of a_1 and a_2 if it exists and gives *fail* otherwise. By abuse of notation we will say that $MGU(a_1,a_2)$ does not exist if it gives value *fail*. We assume that when $MGU(a_1,a_2)$ exists it is equal to the identity function on all variables not occurring in a_1 or a_2.

For the purposes of computation another formulation of this function has been used to great effect in real PROLOG implementations. This is the so-called "structure-sharing" idea [3, 16]. This will be used in the later semantics and gives a similar function defined by

$$MGU_{SS}: Subst \times Atom \times Num \times Atom \rightarrow Subst + \{fail\}$$

$$MGU_{SS}(\phi,a_1,n,a_2) = MGU(\phi(a_1),\Psi_n(a_2))$$

with n acting as renaming index. Moreover, it achieves this effect without performing the substitutions implied by the above definition.

4 SLD-TREES AND THEIR PROPERTIES

4.1 SLD-resolution

We use the terminology of Apt and van Emden [1] and define an *SLD-derivation* to be a sequence $N_0,N_1,\ldots$ of negative clauses such that for each i, if N_i has the form

$$\leftarrow a_1\wedge\ldots\wedge a_n$$

then N_{i+1} has the form

$$\leftarrow\theta(a_1\wedge\ldots\wedge a_{k-1}\wedge\Psi(a'_1\wedge\ldots\wedge a'_m)\wedge a_{k+1}\wedge\ldots\wedge a_n)$$

satisfying

1. $1\leqslant k\leqslant n$ (a_k is the *selected atom*)
2. $a'\leftarrow a'_1\wedge\ldots\wedge a'_m$ is a clause in P (the *selected clause*)
3. Ψ is a renaming such that $\Psi(a'\leftarrow a'_1\wedge\ldots\wedge a'_m)$ has no variables in common with $a_1,\ldots,a_n$
4. $\theta a_k = \theta(\Psi a')$

An *SLD-refutation* of N_0 is an SLD-derivation ending in $\leftarrow$true, the contradictory goal. If $N_0,\ldots,N_{m-1},N_m = \leftarrow$true is an SLD-refutation which uses substitutions $\theta_1,\theta_2,\ldots,\theta_m$, the composition $\theta_m\circ\ldots\circ\theta_1$ is called the *answer substitution*.

4.2 SLD-trees

Given a PROLOG program P;$\leftarrow$q an *SLD-tree* (as defined in [1]) consists of nodes labelled with negative or empty clauses. If node N has label $\leftarrow a_1\wedge\ldots\wedge a_m$ then a selected atom a_k is designated $(1\leqslant k\leqslant n)$. Further N has exactly one son for each clause $a'\leftarrow b'$ such that a_k and a' can be unified (after renaming), the son being labelled with the resolvent. The sons are ordered according to the order that clauses appear in P. The root is labelled $\leftarrow$q.

An SLD-tree represents a collection of SLD-derivations all starting with $\leftarrow$q. A complete search strategy is to traverse an SLD-tree (its branching factor is bounded by the number of clauses) searching for refutations. A parallel search strategy would thus be complete. However, for efficiency reasons, most PROLOG interpreters defined so far have used a depth-first search strategy which is necessarily only weakly complete (i.e. if it terminates then it is complete, but non-termination may occur even when a refutation exists. See [12]). An exception is the Loglisp system of Robinson [17] which uses breadth-first search.

The main impetus of this work has been to describe the standard form of PROLOG interpreter summarily described as leftmost search of the leftmost SLD-tree rather than the full generality of choice of SLD-resolution rules. By the *leftmost SLD-tree* we mean the one in which the leftmost atom is always selected for resolution. Its leftmost search is that implied by the depth-first search of sons of a given node ordered by the textual ordering of the clauses of the program.

5 OPERATIONAL SEMANTICS

In all the interpreters which follow the program is assumed to be of the form P;$\leftarrow$q. We will feel free to reference the sentence P freely from the interpreters - more formally we would add it to each interpreter state for use when required. The start will be specified by q.

5.1 Interpreters without Structure-sharing

The definition of "refutation" in essence defines the following non-deterministic PROLOG interpreter. The interpreter is based on the leftmost SLD-tree given by choosing a_1 as the selected atom in the resolvent $a_1\wedge\ldots\wedge a_n$. We have said nothing yet about the output - this will be added later.

The following interpreter simulates interpreter I by depth-first left-right search of the leftmost SLD-tree, using a stack to maintain backtracking information.

I: Non-deterministic rewriting interpreter

State = Body × Num
Start state: (q,0)
Stop state : (true,n)

State transition rule (→ is a relation on State × State)

$(a \wedge b, n) \rightarrow (\theta\Psi_{n+1}b' \ @\ \theta b, n+1)$
if $(a' \leftarrow b')$ is a clause in P and
$\theta = MGU(a, \Psi_{n+1}a')$ exists

II Rewriting interpreter with backtracking

[Omitted]

5.2 Interpreters with Structure-sharing

The structure sharing technique of [3, 16, 19] may be used to avoid completely with creation of new atoms and terms not present in the original program. (We note that every substitution in the previous interpreters in general creates a different atom when applied.) The technique is widely used in real PROLOG interpreters and represents a substituted body by a pair (shape,θ) where shape is a substructure of the original program together with a substitution θ representing a deferred application. This is also used to represent renaming, and a renamed body will be represented by a pair (body,n) with the index n representing Ψ_n. The reason that this is worthwhile is that there is a version of the unification algorithm (MGU_{SS}) which can perform unification efficiently on two such objects [16]. Applying this idea to interpreter I, we obtain

III Non-deterministic structure-sharing interpreter

σ: State = Stack × Subst × Num
s: Stack = (Body × Num)*
Start State: ((q,0)::nil, Id, 0)
Stop State: (nil, ϕ, n) giving an answer substitution ϕ

Transition rules

1. Select subgoal
$((a \wedge b,m)::s, \phi, n) \rightarrow ((b',n+1)::(b,m)::s, \theta \circ \phi, n+1)$
if $\theta = MGU_{SS}(\phi \circ \Psi_m, a, n+1, a')$ exists and $(a' \leftarrow b')$ is a clause of P

2. Satisfied subgoal
$((true,m)::s, \phi, n) \rightarrow (s,\phi,n)$

A stack $s = [(b_1,m_1),\ldots,(b_D,m_D)]$ represents a conjunction of as-yet-unsatisfied goals, each with its own renaming index. In state (s,ϕ,n) n is the current renaming index (= the number of steps in the current refutation attempt so far), s represents the current goal and ϕ holds the current accumulating answer substitution.

[Omitted: correctness proof.]

We now consider backtracking with structure sharing. This is obtained from interpreter III by using a stack in precisely the same way that interpreter II was obtained from interpreter I. We take advantage of the fact that the length of the State gives the current renaming index.

IV Structure-sharing interpreter with backtracking

σ: State = (Stack × Subst × Clause*)*
s: Stack = (Body × Num)*
Start State = ((q,0)::nil, Id, P) :: nil
Stop State = nil

Transition rules

1. Apply clause to select subgoal
$((a \wedge b,m)::s, \phi, (a' \leftarrow b')::c^*) :: \sigma$
$\rightarrow ((b',n+1)::(b,m)::s, \theta \circ \phi, P) :: \sigma'$
if $\theta = MGU_{SS}(\phi \circ \Psi_m, a, n+1, a')$ exists
$\rightarrow \sigma'$ otherwise
where n = length(σ)
and $\sigma' = ((a\ b,m)::s,\phi,c^*) :: \sigma$

2. Goal not (further) satisfiable: backtrack
$((a \wedge b,m)::s, \phi, nil) :: \sigma \rightarrow \sigma$

3. Satisfied goal; continue with brother goals
$((true,m)::s, \phi, c^*) :: \sigma \rightarrow (s,\phi,P) :: \sigma$

4. Satisfied main goal: produce output (see later) and backtrack
$(nil, \phi, c^*) :: \sigma \rightarrow \sigma$

ϕ is an answer substitution

Implementation remarks: the bodies a∧b, b occurring here are all tails of bodies appearing in the given program and so can be represented by pointers (as can the clause sequences in a similar fashion to interpreter II). The copying of stack s can clearly also be avoided by the use of a pointer, leaving the only significant overhead the manipulation of substitutions. These can also be implemented efficiently using techniques such as those described in [6, 19].

5.3 Handling *cut*

We would now like to add to the definition some description of "cut". Cut is generally described as making the computation, from the call of a particular atom in whose definition the cut symbol appears, deterministic up to that point. Therefore the simple technique adopted here (which corresponds closely to real implementations) is to add a dump component to Stack representing (the backtracking part of) the calltime state which can be restored on encountering the cut, thereby removing backtracking points encountered since the call. One small complication is that the next renaming index (n) may no longer be the length of the state, so we must insert it as an explicit component as in the non-deterministic versions (I and III).

V Structure-sharing interpreter with cut

σ: State = (Stack × Subst × Clause* × Num)*
s: Stack = (Body × Num × Dump)*
d: Dump = State
Start State: ((q,0,nil):: Id, P, 0) :: nil
Stop State : nil

Transition rules

1. Apply clause to select subgoal, saving σ as Dump for cut

$$((a\wedge b,m,d)::s,\ \phi,\ (a'\leftarrow b')::c^*,\ n)\ ::\ \sigma \rightarrow ((b',n+1,\sigma)::(b,m,d)::s,\ \theta\circ\phi,\ P,\ n+1)\ ::\ \sigma'$$

if $\phi = MGU_{SS}(\phi\circ\Psi_m, a, n+1, a')$ exists
→ σ' otherwise
where σ' = ((a∧b,m,d)::s,ϕ,c*,n) :: σ

2. Goal not (further) satisfiable: backtrack

$$((a\wedge b,m,d)::s,\ \phi,\ nil,\ n)\ ::\ \sigma \rightarrow \sigma$$

3. Cut operator: remove part of backtrack stack

$$(("!"\wedge b,m,d)::s,\ \phi,\ c^*,\ n)\ ::\ \sigma \rightarrow ((b,m,d)::s,\ \phi,\ c^*,\ n)\ ::\ d$$

4. Satisfied goal: continue with brother goals

$$((true,m,d)::s,\ \phi,\ c^*,\ n)\ ::\ \sigma \rightarrow (s,\phi,P,n)\ ::\ \sigma$$

5. Satisfied main goal: produce output (see 5.4) and backtrack

$$(nil,\ \phi,\ c^*,\ n)\ ::\ \sigma \rightarrow \sigma$$

ϕ is an answer substitution

Note that the Dump component is always a terminal segment of the current state and so can be implemented as a pointer into it.

6 DOMAINS AND SUBSTITUTION SEQUENCES

Suppose interpreter V (for example) is extended to write its answer substitution each time a transition by rule 5 is taken, there are then three possible results of the computation; production of an infinite sequence of output; production of finitely many outputs followed by termination; or production of finitely many outputs followed by an infinite computation without output. Domain theory as used in denotational semantics provides a suitable formalism for describing these possibilities, and is also used in Section 7.

6.1 Review of Domain Theory

[Details omitted, but follow the lines of [14]. Keypoints:

+ is the coalesced sum
× is the smash product
$D_\perp$ is D augmented by a new least element ⊥]

6.2 Extracting the Result of a Computation

The desired domain of finite or infinite sequences of substitutions can be defined by

$$Subst^+ = \{nil\}_\perp + Subst \times (Subst^+)_\perp$$

To see this, write the pairing operation as :: (a *right-associative* infix CONS operator), and write [] for nil and [a,b,...,z] for a::[b,...,z]. This gives a natural injection of Subst* into $Subst^+$. The definition ensures for example that [a,b,⊥,c,d] = [a,b,⊥], so the only infinite sequences in $Subst^+$ have the form $[a_1,a_2,...]$ where each $a_i \in Subst$. Intuitively, $Subst^+$ is the class of (tail-)lazy substitution lists which can be terminated in nil.

The output is now obtained by defining functions $Output_k$ which give the partial output of a program after k evaluator steps and are defined by

$Output_k$: State → $Subst^+$
$Output_0(\sigma) = \perp$
$Output_{k+1}(nil) = nil$
$Output_{k+1}((nil,\phi,c^*,n)\ ::\ \sigma) = \phi\ ::\ Output_k(\sigma)$ (add Output by rule 5)
$Output_{k+1}(\sigma) = Output_k(\sigma')$ if σ→σ' by rule 1, 2,3 or 4.

The output can now be defined as the limit (k→∞) of $Output_k$(Start-State).

The great virtue of such a definition of the output from a program is that it distinguishes between the output of a program which produces substitutions σ_1,σ_2 and then stops, and one which produces the same substitutions but subsequently carries on computing forever without producing further results. Out model of the output of the former would be $\sigma_1 :: \sigma_2 :: nil$ and the latter $\sigma_1 :: \sigma_2 :: \perp$.

As an aside, we observe that the conventional notion is that this limit should be taken at $\omega = \{0,1,2,...\}$, but that State may then still contain unexplored backtracking choices due to the depth-first search. We leave as an open question whether a generalisation of the notion of sequence to higher ordinals would produce a better (larger) result set. This matter of a SLD-tree traversing interpreter not covering the whole of the search space after ω_0 steps is examined in much more detail in [12].

6.3 A Domain Definition for Cut

An elegant way to handle *'cut'* denotationally can be obtained by slightly modifying the domain equation for $Subst^+$. We first introduce the type operator ∧ which acts on a given type, α say, in the following manner:

$$\alpha^\wedge = \{nil,cut\}_\perp + \alpha \times (\alpha^\wedge)_\perp$$

This defines the type $\alpha^\wedge$ to consist of finite or infinite lists of elements of α, such that the finite lists may be terminated with either *nil* or *cut*. By a convenient abuse of notation we will consider the types α^+ and α^* to be subsets of $\alpha^\wedge$ by the natural inclusions. The full power of $\alpha^\wedge$ will only be needed to describe the semantics of *cut*, elsewhere α^+ can (and will) be used.

Again we define the diadic append operator (written as infix @):

@: $\alpha^\wedge \times \alpha^\wedge \rightarrow \alpha^\wedge$
a: k @ m = a :: (k @ m)
nil @ m = m
cut @ m = cut
⊥ @ m = ⊥

'@' is thus a (continuous) function which acts like the traditional append on lists terminated in *nil*, but ignores the second parameter for those whose first parameter ends in *cut*. Technically this only defines @ on finitely constructed elements of $\alpha^\wedge$

(here these are just finite sequences ended with nil, cut or ⊥). The definition is extended to the whole of $\alpha^\wedge$ (also including the infinite sequences) by defining

$$\ell \;@\; m = \bigsqcup_{\ell' \sqsubseteq \ell,\, m' \sqsubseteq m,\ \ell',m' \text{ finite}} \ell' \;@\; m'$$

Analogous extensions to the whole of $\alpha^\wedge$ are implicitly assumed in the definitions of Append and Uncut given below.

We now define an Append functional (compare the Σ functional for summation) to be a version of @ extended to concatenate a sequence of such objects. If t[i] is a (mathematical) term with free variable i then

$$\text{Append } t[i] = t[a] \;@\; \text{Append}(t[i]) \qquad i \in a::b \qquad\qquad i \in b$$

$$\text{Append } t[i] = x \text{ if } x = \text{nil, cut or } \bot \qquad i \in x$$

For example, if x = [1::nil, nil, 2::(3::cut), 4::(5::nil)] then

$$\text{Append } i = 1::(2::(3::\text{cut})). \qquad i \in x$$

7 DENOTATIONAL VERSION

7.1 A Simple Semantic Definition

Again we develop a series of semantics. The first version will be the simplest definition which has any denotational flavour. We will discuss its shortcomings and derive successively 'better' semantics.

To this end we start by defining a Lookup function which gives a representation of the sons of a node in an SLD-tree:

Lookup: Atom × Sentence → (Body × Subst)*
Lookup(a,[]) = []
Lookup(a,(a'←b')::z) = (b',θ) :: Lookup(a,z) if θ = MGU(a,a') exists
= Lookup(a,z) otherwise

We will ensure that uses of Lookup are such that its two parameters contain disjoint sets of variables.

The very simplest, 'denotational' semantics which can be given for PROLOG is given by:

Semantics I

B: Body → Num → Sentence → Subst → Subst⁺
V: Program → Subst⁺

B [[true]] n P ϕ = [ϕ]
B [[a ∧ b]] n P ϕ = Append (B [[b"]] (n+1) P (θ∘ϕ)) (b',θ) ∈ Lookup(a, Ψ_{n+1}(P))
where b" = θ(b' @ b)

V [[P;←q]] = B [[q]] 0 P Id

This directly models interpreter I, the simple leftmost tree-walking system, presented in Section 5. Nondeterminism is modelled by letting the output produced at a SLD-tree node with sons be merely the concatenation (in order) of the output produced by each of the sons. Note that the sentence P must be explicitly carried around in the greater formality of denotational semantics, compared to the operational semantics.

It is illuminating to consider why this should not be considered a valid form of denotational semantics, since (we claim that) it gives the correct answer substitution list. The main reason is that it fails to match the spirit of a denotational definition since, apart from the meaning of a program being a substitution list, there are no denotations - just symbol manipulation. A general, but unspoken, requirement of denotational definitions is that they should only contain substructure of the original program inside [[ ]] brackets. A subsidiary reason is that the definition does not derive meaning of substructure of the program, but merely follows an interpretive style blindly.

7.2 Lemma

PROLOG implementations typically try to satisfy goal (a∧b) by first satisfying a, yielding an answer substitution ϕ, and then satisfying ϕb. This can be justified as follows.

LEMMA A negative clause $\leftarrow a_1 \wedge \ldots \wedge a_n$ is refutable if and only if for some ϕ, $\leftarrow a_1$ is refutable with answer substitution ϕ and $\phi(a_2 \wedge \ldots \wedge a_n)$ is refutable.

[Proof omitted]

7.3 Better Semantic Definitions

As suggested in the discussion of 7.1 we need a technique like structure-sharing to avoid manipulation of program text. Following an idea from interpreter III, it is natural to use two semantic functions

A: Atom → Num → Sentence → Subst → Subst⁺
B: Body → Num → Sentence → Subst → Subst⁺

and to define B [[b]] m P ϕ as the sequence of all refutations of $(\phi \circ \Psi_m)$b obtained under PROLOG's search strategy, and define A analogously. The lemma just proved gives a way to compute B [[a∧b]] which can be informally described by

B [[a∧b]] m P ϕ = let [θ1,θ2,...] = A [[a]] (m+1) P ϕ in (B [[b]] m P(θ1∘ϕ)) @ (B [[b]] m P(θ2∘ϕ)) @ ...

A lazy implementation of this corresponds to PROLOG's usual search strategy. There is, however, a technical problem to solve before giving a structure-sharing denotational semantics based on this idea. The complication is that we need to know the highest renaming index used inside A [[a]] , in order to avoid conflicts with possible renaming needed for atoms in b (or their descendants). Therefore the domain must be complicated somewhat:

A: Atom →Num →Sentence →Subst→ Num →(Subst × Num)⁺.

To use this in the semantics an auxiliary function to map down a sequence of pairs extracting the sequence of first components is required:

First: (Subst × Num)⁺ × Subst⁺
First ⊥ = ⊥
First [] = []
First ((ϕ,n) :: M) = ϕ :: First(M)

This enables us to write

Semantics II

A: Atom → Num → Sentence → Subst → Num → (Subst × Num)$^+$
B: Body → Num → Sentence → Subst → Num → (Subst × Num)$^+$
V: Program → Subst$^+$

A [[a]] m P ϕ n = Append(B [[b]] (n+1) P ($\theta \circ \phi$) (n+1))
(b,θ) $\in$ Lookup((($\phi \circ \Psi_m$)a, Ψ_{n+1}P)

B [[true]] m P ϕ n = [(ϕ,n)]
B [[a$\wedge$b]] m P ϕ n = Append(B [[b]] m P ($\theta \circ \phi$) n')
(θ,n') $\in$ A [[a]] m P ϕ n

V [[P;$\leftarrow$q]] = First (B [[q]] 0 P Id 0)

Semantics II still fails to meet the spirit of a denotational definition due to the manipulation of P as a textual object. Worse still, this leads to the use (in A) of a semantic function to a body which is not a substructure of the rule being defined. This can lead to an infinite regress in a *mathematical* definition which is not permissible, unlike the situation in programs which the mathematics describes.

This objection may be overcome by representing the sentence by a recursively defined environment Env which replaces Sentence in the semantic equations and represents the effect of the sentence in terms of its effect of mapping atoms to substitution lists. We have essentially used the definition of Env to replace the symbol string representation of Sentence with its meaning. This leads to:

Semantics III

ρ: Env = Atom → Num → Subst → Num → (Subst × Num)$^+$
B: Body → Num → Env → Subst → Num → (Subst × Num)$^+$
D: Sentence → Env → Env
V: Program → Subst$^+$

B [[true]] m ρ ϕ n = [(ϕ,n)]
B [[a$\wedge$b]] m ρ ϕ n = Append(B [[b]] m ρ ($\theta \circ \phi$) n')
(θ,n') $\in$ ρ[[a]] m ϕ n

D [[[]]] ρ a m ϕ n = [ ]
D [[a'$\leftarrow$b'::c*]] ρ a m ϕ n = σ @ σ'
where σ = B [[b']] (n+1) ρ ($\theta \circ \phi$) (n+i)
if θ = MGU$_{SS}$($\phi \circ \Psi_m$,a,n+1,a') exists
= [] otherwise
and σ' = D [[c*]] ρ a m ϕ n

V [[P;$\leftarrow$q]] = First (B [[q]] 0 ρ Id 0)
where ρ = fix D [[P]]

The last line is well defined since D [[P]] is a continuous function on a cpo.

Now every semantic function is applied only to substructures of its syntactic argument. Lookup has been replaced by ρ [[a]] m ϕ n, which according to the definition of D finds all possible ways to satisby a by resolution with the clauses of P, taken in left-to-right order.

The remaining problem is to define a semantics for cut. The existence of the cut symbol poses some problems for a direct denotational semantics for PROLOG. This is normally seen as removing some substitutions from the meaning of a program in a continuation-style semantic model. In Section 6.1 we modelled *cut* directly by introducing a data-type $\alpha^\wedge$ analogous to lists but with an extra constructor, cut, which as a special (absorbing) effect on the append operator, and has previously been unused in the semantic equations.

We require the auxiliary function

Uncut: $\alpha^\wedge \rightarrow \alpha^+$
Uncut(a::k) = a :: Uncut(k)
Uncut(cut) = nil
Uncut(nil) = nil

which has the effect of replacing any terminal *cut* in its parameter with *nil*. This is used to limit the effect of a cut to the bodies of clauses which match the subgoal invocation.

7.4 Semantics for PROLOG with cut

Our final semantics (which we believe formalises the standard semantics for depth-first PROLOG with cut) is:

ρ: Env = Atom → Num → Subst → Num → (Subst × Num)$^+$
B: Body → Num → Env → Subst → Num → (Subst × Num)$^\wedge$
D: Sentence → Env → Env
V: Program → Subst$^+$

B [[true]] m ρ ϕ n = [(ϕ,n)]
B [[a$\wedge$b]] m ρ ϕ n = Append(B [[b ρ m]] ($\theta \circ \phi$) n')
(θ,n') $\in$ ρ [[a]] m ϕ n
B [["!"$\wedge$b]] m ρ ϕ n = (B [[b]] m ρ ϕ n) @ cut

D [[[]]] ρ a m ϕ n = []
D [[a'$\leftarrow$b'::c*]] ρ a m ϕ n = Uncut(σ @ σ')
where σ = B [[b']] (n+1) ρ ($\theta \circ \phi$) (n+1)
if θ = MGU$_{SS}$($\phi \circ \Psi_m$,a,n+1,a') exists
= [] otherwise
and σ' = D [[c*]] ρ a m ϕ n

V [[P;$\leftarrow$q]] = First (B [[q]] 0 ρ Id 0) where ρ = fix D [[P]]

The above definition has the property that substitutions are merely composed and that no substitution (or renaming) is ever applied to any part of the program source. Indeed, substitutions are never applied, except implicitly during the unification process (MGU$_{SS}$). This gives the property that the only terms which appear as the first parameter to any semantic function (B,D or V) are subterms (in the mathematical sense) of the original program [[P;$\leftarrow$q]] .

This is an essential requirement for being able to base a compiler on the semantic definitions given here. That this is so, is a consequence of the fact that the given program has only a finite number of subterms, and we can compile each one (actually only a subset of these as indicated by the semantic equations) into code by generating a fragment of code for B [[x]] for each terminal segment x of (the Body of) each Horn clause in the program. More details on compiler generation according to this scheme are given by Christiansen and Jones [5]. Work is currently in progress to develop a real PROLOG compiler by such techniques.

8 CONCLUSIONS

We have given operational and denotational definitions of the PROLOG language including the *cut* operation. These we developed by stepwise refine-

ment from a mathematical formulation of computation step. We believe that technique is a valid method of deriving semantics and is much easier to understand than the traditional approach of just giving semantic equations with cursory explanation. In a sense we 'show our working' involved in producing an elaborate semantics. That this technique is applicable follows from the fact that PROLOG is based on an already well-understood theory of computation step, a fact which is unfortunately not true of most programming languages.

It would be interesting to repeat the work for the equivalent case of a high-level language soundly based on λ-calculus.

9 REFERENCES

[1] Apt, K.R. and van Emden, M.H., Contributions to the theory of logic programming. *Journal of the ACM* 29(3):841-862, 1982.

[2] Björner, D., Jones, C.B., *The Vienna Development Method: the meta-language*. Springer-Verlag Lecture Notes in Computer Science, 1978.

[3] Boyer, R.S. and Moore, J.S., The sharing of structure in theorem-proving programs. In Meltzer, B. and Michie, D. (editors), *Machine Intelligence 7*. Edinburgh University Press, 1972.

[4] Burstall, R.M. and Darlington, J. A Transformation System for Developing Recursive Programs. *Journal of the ACM* 24(1):44-67, January, 1977.

[5] Christiansen, H., and Jones, N.D., Control flow treatment in a simple semantics-directed compiler generator. In *Proc. IFIP WG2.2 working conference on math. found. comp. sci.*, North-Holland, 1983.

[6] van Emden, M.H., An interpreting algorithm for PROLOG programs. In *Proc. 1st Intl. Logic Programming Conf.*, Marseille, 1983.

[7] Komorowski, H.J., Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of PROLOG. In *Proc. 9th ACM Symp. on Principles of Programming Languages*, 1982.

[8] Kowalski, R., Predicate Logic as a Programming Language. In *Proc. IFIP congress*. North-Holland, Stockholm, 1974.

[9] Kowalski, R., Algorithm = Logic + Control. *Communications of the ACM* 22(7), 1979.

[10] Landin, P.J., The mechanical evaluation of expressions. *Computer Journal S*, 1964.

[11] Manna, Z. and Waldinger, R., Deductive synthesis of the unification algorithm. *Science of Computer Programming* 1(1), 1981.

[12] Mycroft, A., Logic Programs and many-valued logic. In *Proc. Symposium on Theoretical Aspects of Computer Science*, Paris, 1984. Springer-Verlag LNCS.

[13] Nilsson, J.F., *Specification and development of a prolog system*. Technical Report IDS60, Tech. Univ. of Denmark, Lyngby, 1981.

[14] Plotkin, G., The Category of Complete Partial Orders: A Tool for Making Meanings. *Proc. Summer School on Found. Art. Intel, and Comp. Sci.*, Instituto di Scienze dell'Informazione, Universita di Pisa, 1978.

[15] Robinson, J.A., A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12(1), 1965.

[16] Robinson, J.A., Computational logic: the unification computation. In Meltzer, B. and Michie, D. (editor), *Machine Intelligence* 6. Edinburgh University Press, 1971.

[17] Robinson, J.A. and Sibert, E.E., *LOGLISP - an alternative to PROLOG*. Technical Report 7-80, School of Comp. and Info. Sci., Syracuse University, 1980.

[18] Stoy, J., *Denotational semantics*. MIT press, 1977.

[19] Warren, D.H.D., *Implementing PROLOG - compiling predicate logic programs*. DAI research report 33-40, Dept. of Artificial Intelligence, Edinburgh University, 1977.