OPERATIONAL AND DENOTATIONAL SEMANTICS OF PROLOG

BIJAN ARBAB AND DANIEL M. BERRY*†

A Vienna Definition Language operational semantics of PROLOG, which includes the cut, the database, and the extra-logical operations, is presented. This semantics serves as the basis for deriving a denotational-continuation-style semantics of PROLOG through a systematic transformation of the operational semantics by a method described by Berry.

1. INTRODUCTION

The features that make the language PROLOG interesting to define formally are the refutation procedure, its use as a deterministic programming language, the backtracking, the cut, the database, and the extra-logical features. The refutation procedure of PROLOG is based on Horn clauses [9]. A formal characterization of the latter can be used as a formal definition of the former. A number of precise characterizations of semantics of Horn clauses have been discussed by Kowalski [9], van Emden and Kowalski [6], and Apt and van Emden [1]. These characterizations, however, do not capture the semantics of PROLOG as a deterministic programming language. In an attempt to remedy this situation, Jones and Mycroft [8] give an operational and a denotational semantics of PROLOG as a deterministic programming language. Both of these semantics are based on Kowalski's SLD refutation procedure [9]. These semantics include the cut as well as the basic goal searching. They do not, however, include the database and extra-logical operations. A still more recent attempt, by J. F. Nilsson [12], gives a progression of Vienna Definition Language operational semantics.
Method (VDM) descriptions of a variety of PROLOG implementations, but also does not deal with the cut and the extra-logical operations.

This paper describes an operational and a denotational semantics of PROLOG including the cut, the database, and the extra-logical facilities. These semantics differ from those of Jones and Mycroft in that they are not based on the SLD-refutation procedure. This procedure tends not to be well-defined when the underlying program driving the refutation is allowed to change during the refutation. Also, the various characterizations of logic programs developed in [9, 6, 1], which include SLD refutations, simply do not apply to perpetual processes, i.e., nonterminating processes that do useful work. See Lloyd [10] for a discussion and an outline of a solution. Both of the these features have been defined in other guises for other languages using operational formalisms, such as abstract interpreters.

Ultimately, all of the above features are formally definable; they are all implemented and can at the very least be defined by one such implementation. Of course, it may be desired to give a more abstract definition of the language. The problem with taking an implementation as the definition is that other equally used implementations may differ from it. However in the PROLOG community, one implementation, the DEC 10 interpreter, seems to be accepted as the standard. That implementation serves as the basis for the present operational semantics of the full language.

The starting point is Vienna Definition Language (VDL) [11] operational semantics of PROLOG. Since the real PROLOG is in fact very procedural, it is proper to start with a procedural definition, as a VDL definition must be. This VDL definition is an interpretive definition that closely follows working interpreters of PROLOG written in PROLOG, one produced by Nilsson [13] and the other by Parker and Eggert [14]. This close modelling increases the confidence in the correctness of the VDL definition. The major difference between the VDL definition and those of Nilsson and of Parker and Eggert is that the VDL definition makes the backtracking mechanism explicit, while the latter two do not (because they use the built-in backtracking of the language, PROLOG, in which the interpreter is written). The VDL semantics is then used as the basis for systematically deriving a denotational continuation semantics for PROLOG. Just as in [8], first an operational semantics was produced, and then this was transformed systematically into the desired denotational continuation semantics.

The method used to derive the denotational from the VDL semantics is of interest also. The method used is that suggested by Berry in [3] in which it is shown

1. how to convert an arbitrary deterministic VDL definition of a language, expressed in denotational normal form, into a denotational continuation semantics of the same language and
2. how to convert an arbitrary nondeterministic VDL definition of a language, expressed in denotational normal form, into a denotational semantics of the same language.

The application of the method to produce the desired denotational semantics of PROLOG consists of two steps:

1. Produce a deterministic denotational normal form VDL definition of PROLOG based on the PROLOG interpreter of PROLOG.
(2) Apply the transformation of [3] to produce the desired denotational continuation semantics of PROLOG.

The reader is assumed to be familiar with the basics and the notation of VDL, PROLOG, and denotational semantics. The references [11, 16, 5, 7] provide good background on all of these topics.

2. VDL DEFINITION OF PROLOG

The VDL definition of PROLOG is given in deterministic, denotational normal form. A VDL definition is deterministic if no forks are ever introduced into the control tree.

A VDL definition is in denotational normal form (DNF) if the state consists only of at most three components that can be identified as the control tree, the storage, and the environment. The control tree must be present, while the other two are optional. The control tree is actually a single subobject of the state object and is as defined for standard VDL in [11]. The other two components may each consist of a number of subobjects considered together to build the component. The environment is some mapping from program identifiers to some sort of location names. The storage is some mapping from these storage location names to some sort of values. As mentioned, these are optional; in fact, in many cases, the environment is absorbed into the control tree as an explicit argument to each instruction.

Because of what each of the environment and the storage map, the environment is procedure-local while the storage is global. The environment is changed with each procedure invocation and return while the storage is unchanged at these times. During the execution of the procedure body, the environment is unchanged while the storage may change.

It should be clear that any VDL definition of a programming language can be put in denotational normal form, as putting it in this form requires a reorganization of the state with no loss of information. Making a VDL definition deterministic does involve a loss of information as all but one of the possible computational histories are excluded. However, except as used to model parallelism, nondeterminism in a definition is not considered essential; it is used mostly to allow arbitrary orders of evaluation of operands of expressions.

In practice, it is easy to write a deterministic VDL definition in DNF from the beginning. In the case of the PROLOG machine, it was observed that the current level and the mapping of identifiers to level-named term-level pairs constitute the environment because they change for each procedure invocation and return. It was decided to let the two environment objects be arguments to the instructions. It was observed that the database, the database pointer, the output, and the backtrack point stack all constitute the storage of the PROLOG machine, since they remain unchanged at procedure invocation and return but get modified during the PROLOG analogs to assignment, adding and retracting axioms, saving backtrack points, and backtracking.

The VDL definition given below consists of an abstract syntax describing the programs and the states and a collection of instructions whose job is to execute the program. Notice that the syntax of PROLOG assumes the existence of disjoint sets of symbols for predicates and variables but not functions, as most implementations do not provide the latter. The form of the database is also of interest. The database is a mapping from
predicate names, atoms, to clauses since most implementations of PROLOG provide some kind of indexing on at least the predicate names of the clauses in the database. The renaming of variables is handled in the usual manner, i.e., a unique name is carried along, and variables are indexed according to this unique name. Seek is the top level instruction, and it determines if the goal has been achieved or not; it also checks to see if more than one answer is to be constructed. The cut, database operations, and extra-logical facilities are handled by this instruction as well. The database-seek instruction is the heart of the interpreter and it searches the database for a clause whose head matches the current goal and calls the seek instruction with the new goals due to the body of the selected clause.

See Figure 1 for a picture of the structure of the goal list that is presented as the program to seek to evaluate.

![Figure 1: Representation of Program](image)

The VDL definition of PROLOG follows. The definition itself is in sans-serif fonts, and the commentary, which precedes the relevant portion of the definition, is in normal, avec-serif fonts:

Denotational Normal Form of the VDL Definition of PROLOG:
Syntax of PROLOG Programs described in VDL:

\[
\text{is-program} = \text{is-goal-list}
\]

\text{Is}-\text{goal} also describes any goal occurring during execution, not just those occurring as the top-level goal requested by a user. The \text{s-level} component is used by the interpreter to rename the variables of goal clauses in order to distinguish them from the variables of the stored clauses. This renaming process is necessary in order to avoid name clashes between variables of the goal clauses and those of the stored clauses. For example, in order to properly unify the goal \( a(x) \) with the stored clause \( a(x) : - b(x) \), the \( x \) in the goal clause should be renamed to, say, \( x1 \).

\[
\text{is-goal} = (\langle \text{s-body-list:is-body-list}, \langle \text{s-level:is-n} \rangle \rangle)
\]

\[
\text{is-body} = \text{is-term} \lor \text{is-cut} \lor \text{is-database-op}
\]

Because \text{is-var} is one of the alternatives for \text{is-term}, a variable is allowed to be a term that can get evaluated. This option in effect provides the meta-evaluation operator, which is present in all PROLOG interpreters. Without it, clauses of the form \( \text{not}(x) : - x, !, \text{fail} \) would not even be considered well-formed.

\[
\text{is-term} = \text{is-var} \lor \text{is-pred}
\]

\[
\wedge \text{is-var} = \text{set of variable identifiers}
\]

\[
\text{is-pred} = (\langle \text{s-pname:is-atom}, \langle \text{s-args:is-term-list} \rangle \rangle)
\]

\[
\wedge \text{is-atom} = \text{set of atoms}
\]

This definition allows cuts! The Jones and Mycroft definition allows cuts also, but the run-time semantics provided for them cannot be correctly generalized for use with the full PROLOG [8]. Specifically, their definition generates all possible solutions to all goals as if the cut were not present, and then these are pared down to what would remain after the cut. If, as in their definition, no operation has side-effects, then the final result appears correct. However, in the presence of the full language with operations that have side-effects, the extra evaluations can cause side-effects that simply do not occur when the cut is performed properly. Indeed, the purpose of the cut is to avoid these extra evaluations and their side-effects. As mentioned, J.F. Nilsson’s definition does not allow cuts [12].

\[
\text{is-cut} = \{!\}
\]

\[
\text{is-database-op} = (\langle \text{s-name:is-op-name}, \langle \text{s-args:is-clause} \rangle \rangle)
\]

\[
\wedge \text{is-op-name} = \{\text{ASSERT, RETRACT}\}
\]

\[
\text{is-clause} = (\langle \text{s head:is pred}, \langle \text{s-body-list:is-body-list} \rangle \rangle)
\]
is-n = set of integers greater than equal to 0

Syntax of (run-time) State described in VDL: The control component of the state, selected by s-c and satisfying the predicate is-c is as defined by Lucas and Walk [11]. For this definition, however, it is completely deterministic. The s-output component records the bindings for those variables appearing in the top-level goal. The s-db component is the database in which the program clauses are stored. In addition, this component feels the effect of the ASSERT and RETRACT operators as well. The s-dbptr component is a pointer into the database. It always points to the next clause in the database which is to be tried upon a backtracking. The s-btps component is a stack of backtrack points, each consisting of a control component, a database pointer, a level number, and the rest of the stack. Upon a backtracking, the state is reset from the components of the top-most point on the stack. The cut, then, is implemented by popping this stack to the proper level, found by searching for the level whose s-level component matches cut's argument.

is-state = (es-c:is-c>,
<s-output:is-output>,
<s-db:is-database>,
<s-dbptr:is-database-pointer>,
<s-btps:is-backtrack-point-stack>)

is-backtrack-point-stack =
(s-c:is-c>,
<s-dbptr:is-database-pointer>,
<s-level:is-n>,
<s-btps:is-backtrack-point-stack>)

The database pointer consists of an index and a predicate name, which is an atom.

is-database-pointer =
(is-n:is-n>,<s-pname:is-atom>)

The clauses stored in the database are selected, i.e., indexed, by their predicate names. This organization reflects the common practice of hashing the database clauses by their predicate names. Some implementations also place a secondary index based on the first arguments of the predicates. This method could easily reflected in the current definition by a suitable modification of the is-database structure. Note that the atoms selecting the clause list in a database are the same as the predicate names defined by the selected clause list.

is-database = {<a:is-clause-list> | is-atom(a)}

is-output = {<v:is-term> | is-is-var(v)}

is-c = (as in Lucas and Walk)

The environment, as usual, keeps the current bindings of variables.
\[
\text{is-env} = \{ \langle v: \text{is-binding} \rangle \mid \text{is-var}(v) \}\]

\[
\text{is-binding} = \{ \langle n: \text{is-pair} \rangle \mid \text{is-n}(n) \}
\]

\[
\text{is-pair} = (\langle s:\text{-term}:\text{is-term} \rangle, \langle s:\text{-level}:\text{is-n} \rangle)
\]

**Abbreviations:**

- \(\text{DBPTR}\)
- \(\text{DB}\)
- \(\text{BTPS}\)
- \(\text{OUTPUT}\)
- \(\text{C}\)

\[
\begin{align*}
\text{DBPTR} &= s-\text{dbptr}(\xi) \\
\text{DB} &= s-\text{db}(\xi) \\
\text{BTPS} &= s-\text{btps}(\xi) \\
\text{OUTPUT} &= s-\text{output}(\xi) \\
\text{C} &= s-c(\xi)
\end{align*}
\]

**Initial State:** The initial state consists of a database initialized to some unspecified set of clauses and a control poised to seek solutions for the list of goals making up the program, assuming level zero and an empty environment.

\[
\text{is-initial-state}(x) = \langle \exists \text{ program} \rangle (\text{is-goals-list}(\text{program}) \& x = h(\langle \text{s-db:is-database} \rangle, \langle s-\text{c:seek} \rangle(\text{program}, 0, \Omega)))
\]

**Semantic Instructions:** Seek is the main instruction for solving a top-level goal list at some level \(n\) in some some environment. If the top-level goal list is empty, then all the relevant bindings, if any, that have been computed are printed. Otherwise, seek uses goals which are components or descendents of the top-level goal list. If the first conjunct of a conjunctive goal list is empty, i.e., the conjunct has been solved, then proceed to solve the remaining conjuncts. If the goal is a cut, then the stack of backtrack points is popped to the appropriate level. This ensures that backtracking is suppressed for this goal and will resume only at the appropriate level. The data base operations are handled simply by adding or deleting clauses in the \(s-db\) component of the state. If the goal to be solved is a variable, then seek is called recursively on the value of the variable. Note that without this recursion, clauses of the form \(p(x):\ldots, x, \ldots\) would not have any meaning, since the occurrence of the variable \(x\) in the right hand side of the rule would be meaningless. Finally, if none of the above conditions are true, it must be that the goal is a term. In this case, the database is searched to find a set of clauses each of whose clause head name is the same as that of the goal. In order to do this search, the database pointer is initialized so that it is pointing to the first place in the database at which a set of such clauses can be found. The actual search is then performed by \textbf{database-seek}.

1. \textbf{seek}(goals, n, env) =
   \[\text{is} = \\
     \text{print-bindings}(\text{env})\]
   \[\text{is} = \\
     \text{seek}(\text{tail}(\text{goals}), \text{n}, \text{env})\]
   \[\text{is} = \text{cut}(\text{body1})\]
   \[\text{cut-stack-to-level}(\text{level1})\]
   \[\text{is-database-op}(\text{body1})\]
   \[\text{do-database-op}(\text{body1})\]
is-var(body1) →
    seek(var-deref,n,env)
T →
database-seek(firstgoal,restgoals,n,env)
    set-database-pointer(firstgoal)

Where:

goal1 = elem(1)(goals)
level1 = s-level(goal1)
bodylist1 = s-body-list(goal1)
body1 = elem(1)(bodylist1)
firstgoal = μ₀(<s-body-list:body1>,<s-level:level1>)
restgoals = δ(goals;elem(1)-s-body-list-elem(1))
var-deref = lookup(μ₀(<s-term:body1>,<s-level:level1>))

For: is-goal-list(goals) & is-n(n) & is-env(env)

Recall that a database pointer consists of an index and a predicate name. 
**Set-database-pointer** simply sets these components of the global database pointer from its argument goal.

2. set-database-pointer(goal) =
   s-dbptr:μ₀(<s-n:1>,<s-pname:s-pname-elem(1)(goal)>)

For: is-goal(goal) & is-pred(elem(1)-s-body-list(goal)) &
    length(s-body-list(goals))=1

**Print-bindings** extracts and prints from its argument environment, only the values of variables that appear in the top-level goal.

3. print-bindings(env) =
   is-Ω(env)→null
T →
    print-bindings(env1);
    env1:print-term-and-remove-var(var,env)

Where:

-is-Ω(0-var(env))

For: is-env(env)

4. print-term-and-remove-var(var,env) =
   PASS:δ(env,var)
   s-output:μ(OUTPUT;<var:value(var,0,env)>)

For: is-var(var) & is-env(env)
5. value(var,n,env) = value-pair(n=var(env),env)

For: is-var(var) & is-n(n) & is-env(env)

6. value-pair(pair,env) =
   is-var(s-term(pair))→
   value(s-term(pair),s-level(pair),env)
   is-pred(s-term(pair))→
   is-<(s-args(s-term(pair)))→
   \( \mu_0(<s-pname:s-name:s-term(pair)>) \)
   \( T \rightarrow \mu_0(<s-pname:s-name:s-term(pair)>), \)
   \( \mu_0([<elem(i):value(elem(i):s-args:s-term(pair))), \)
   s-level(pair),env>| |1\leq i \leq length(s-args:s-term(pair))>) \)

For: is-pair(pair) & is-env(env)

**Database-seek** searches for the first argument goal in the database. If it succeeds, it unifies the head of the found clause with the that goal. If, in addition this unification succeeds, then the bindings are recorded in the argument environment so that these bindings can be propagated to the second argument goal list. The third argument level number is used in setting backtrack points, in the propagation, and in seeking the body of the clause that was unified with the goal. If the number of clauses left in the database for goal is zero, then a failure occurs, since there are no more clauses left to be tried. If this number is one then there is no need to save a backtrack point on the stack, otherwise one will be pushed into the stack. In either case, an attempt is made to unify the goal with the head of the clause. If this attempt is successful, the goal is replaced by the body of the found clause modified with the appropriate variable bindings, and seek is called with this new set of goals. Otherwise a failure occurs. Then, if there are any more clauses whose names are the same as that of the goal, they will be tried.

7. database-seek(goal,goals,n,env) =
   number-of-clauses = 0 →
   fail
   number-of-clauses = 1 →
   seek(newgoals,n+1,newenv);
   newgoals:build-new-goals(goals,n);
   newenv:unify(clausehead,pair,env);
   clausehead:get-clause-head(n)
   \( T \rightarrow \)
   seek(newgoals,n+1,newenv);
   newgoals:build-new-goals(goals,n);
   newenv:unify(clausehead,pair,env);
   clausehead:get-clause-head(n);
   set-backtrack-point-stack(n)
Where:
  number-of-clauses = length(s-pname(DBPTR)(DB))
  pair = \mu_0(\langle s\text{-term}:s\text{-head}(\text{clause}), s\text{-level}:n \rangle)

For: is-goal(goal) & is-goal-list(goals) & is-n(n) & is-env(env)

8. \texttt{get-clause-head}(n) = \texttt{PASS};\mu_0(\langle s\text{-term}:s\text{-head}(\text{clause}), s\text{-level}:n \rangle)

Where:
  clause = s-n(DBPTR)\circ(s-pname(DBPTR)(DB))

For: is-n(n)

9. \texttt{build-new-goals}(goals, n) = \texttt{PASS};\langle goal\rangle^\text{goals}

Where:
  clause = s-n(DBPTR)\circ(s-pname(DBPTR)(DB))
  goal = \mu_0(\langle s\text{-body-list}:s\text{-body-list}(\text{clause}), s\text{-level}:n \rangle)

For: is-goal-list(goals) & is-n(n)

\textbf{Set-backtrack-point-stack} gathers all the information needed to continue execution in the event of a backtracking and pushes it into the top of the backtrack point stack. The S-C component of a backtrack point specifies the control to be used when continuing. The level number of the point is taken from the argument of the instruction. The database pointer is set to point to the next clause in the database to be tried when continuing. Note, however, that the database itself is a global component and backtracking does not effect it. Recently, there have been some proposals for database operations whose effect would be undone upon backtracking. The semantics of such operations can easily be described by adding a database component to each backtrack point in the stack and saving the current database, or that part to be restored, in the new component.

10. \texttt{set-backtrack-point-stack}(n) = s-btps:
    \mu_0(\langle s\text{-c}:C, s\text{-dbptr}:\mu_0(\langle s\text{-n}:n, s\text{-n(DBPTR)}+1\rangle, s\text{-pname}:s\text{-pname(DBPTR)}), s\text{-level}:n, s\text{-btps}:BTPS \rangle)

For: is-n(n)

\textbf{Fail} either cuts back the backtrack point stack or continues with the next clause in the database. If there is no clause in the database whose predicate name matches that of the
goal, then another failure is forced so that the backtracking will continue at a different level. However, if there are matching clauses in the data base, then execution continues according to the control specified by the top of backtrack point stack.

11. \texttt{fail} =
   \begin{align*}
   \text{is-}\Omega(\text{clause}) & \rightarrow \\
   \text{s-btps:s-btps(BTPS)} & \\
   \text{s-c:fail} &  \\
   \end{align*}
   \begin{align*}
   & \rightarrow \\
   \text{s-c:s-c(BTPS)} & \\
   \text{s-dbptr:\mu_0(<s-pname:s-pname(s-dbptr(BTPS))>,} & \\
   <s-n:s-n\text{p-dbptr(BTPS)})> & \\
   \end{align*}

Where:
\begin{align*}
\text{clause} = s-n(s-dbptr(BTPS))\ast(s-pname(s-dbptr(BTPS)))(DB)
\end{align*}

**Cut-stack-to-level** pops the backtrack stack to the level specified by the argument. It is called when a cut is encountered in order to make sure that computation continues only at the appropriate level.

12. \texttt{cut-stack-to-level}(n) =
   \begin{align*}
   \text{s-level(BTPS)} = n & \rightarrow \\
   \text{pop-backtrack-point-stack} & \\
   \end{align*}
   \begin{align*}
   & \rightarrow \\
   \text{cut-stack-to-level}(n); & \\
   \text{pop-backtrack-point-stack} & \\
   \end{align*}

For: \texttt{is-n(n)}

13. \texttt{pop-backtrack-point-stack} = s-btps:s-btps(BTPS)

The \texttt{unify} procedure succeeds if the first argument \texttt{t1} can be unified with the second argument \texttt{t2}, and if so, it returns the appropriate bindings found in environment argument \texttt{env}. Otherwise, a failure occurs.

14. \texttt{unify}(t1, t2, env) =
   \begin{align*}
   \text{is-}\Omega(\text{lookup}(t1, env)) & \rightarrow \\
   \text{unify}(\text{lookup}(t1, env), t2, env) & \\
   \text{is-}\Omega(\text{lookup}(t2, env)) & \rightarrow \\
   \text{unify}(t1, \text{lookup}(t2, env), env) & \\
   \text{is-var(s-term(t1))} & \rightarrow \\
   \text{PASS}:\mu(\text{env};<s-level(t1)\ast s-term(t1):t2>) & \\
   \text{is-var(s-term(t2))} & \rightarrow \\
   \text{PASS}:\mu(\text{env};<s-level(t2)\ast s-term(t2):t1>) & \\
   \text{functor(t1) = functor(t2) & arity(t1) = arity(t2)} & \rightarrow \\
   \end{align*}
\textbf{unify-args}(arity(t1), t1, t2, env)\\
\quad T \rightarrow \\
\quad \text{fail}\\
\text{For: is-pair}(t1) \land \text{is-pair}(t2) \land \text{is-env}(env)\\

15. \text{functor}(t) = s-pname(s-term(t))\\
\text{For: is-pair}(t)\\

16. \text{arity}(t) = \text{length}(s-args(s-term(t)))\\
\text{For: is-pair}(t)\\

17. \text{lookup}(t, env) = s-level(t) \cdot s-term(t)(env)\\
\text{For: is-pair}(t) \land \text{is-env}(env)\\

18. \textbf{unify-args}(r, t1, t2, env) = \\
\quad r = 0 \rightarrow \\
\quad \text{PASS:env} \\
\quad T \rightarrow \\
\quad \textbf{unify-args}(r-1, t1, t2, newenv); \\
\quad \text{newenv:unify(arg1, arg2, env).}\\
\text{Where:} \\
\quad \text{arg1} = \mu_{0}(<s-term:elem(r)(s-args(s-term(t1))), <s-level:s-level(t1)>) \\
\quad \text{arg2} = \mu_{0}(<s-term:elem(r)(s-args(s-term(t2))), <s-level:s-level(t2)>) \\
\text{For: is-n(r) \land \text{is-pair}(t1) \land \text{is-pair}(t2) \land \text{is-env}(env)}\\

19. \textbf{do-database-op}(dbop) = \\
\quad s-name(dbop) = \text{ASSERT} \rightarrow \\
\quad s-db.:\mu(DB; <p-name:clause-list+>) \\
\quad s-name(dbop) = \text{RETRACT} \rightarrow \\
\quad s-db.:\mu(DB; <p-name:clause-list->) \\
\text{Where:} \\
\quad \text{clause-list+} = p-name(DB)^*<\text{clause}> \\
\quad \text{clause-list-} = \text{delete}(p-name(DB), \text{clause}) \\
\quad \text{delete}(l, e) = (is-list(l) \rightarrow \\
\quad (l, \exists i)(1 \leq i \leq \text{length}(l) \Rightarrow \text{elem}(i, l) = e) \rightarrow l) \\
\quad T \rightarrow (\text{elem}(i, l) = e \rightarrow \ldots)
\[
\mu_0(\{\langle \text{elem}(j) ; \text{elem}(j,l) \rangle | 1 \leq j \leq i-1\})^\ast \\
\mu_0(\{\langle \text{elem}(j) ; \text{elem}(j,l) \rangle | i+1 \leq j \leq \text{length}(l)\}))
\]

\[p\text{-name} = s\text{-pname}(s\text{-head}(\text{clause}))\\
\text{clause} = s\text{-args}(\text{dbop})\]

For: is-value database-op(dbop)

3. DENOTATIONAL DEFINITION OF PROLOG

In converting the deterministic DNF VDL definition into a denotational definition, it was decided to move each list of instruction arguments out of the argument list of the instruction whose semantics is given to the argument list of the meaning of an argumentless instruction. This means that there are only a finite number of objects to which to give meaning.

Thus the signature of the continuation \(C \in \text{is-value} \rightarrow \text{is-database-pointer} \rightarrow \text{is-database} \rightarrow \text{is-output} \rightarrow \text{is-backtrack-point-stack} \rightarrow \text{is-output}\)

and the signature of the instruction meaning function \(M\) is

\[M \in \text{is-inst} \rightarrow \text{is-value}^* \rightarrow \text{is-database-pointer} \rightarrow \text{is-database} \rightarrow \text{is-backtrack-point-stack} \rightarrow \text{is-output} \rightarrow \text{is-c}\]

where the list of values that form the first arguments of meaning of an instruction is determined by the argument list of the corresponding VDL instruction scheme. For example, the instruction scheme for database-seek in the VDL definition has four arguments, a goal, a goal list, a level, and an environment. Thus, for \(M[\text{database-seek}]\), the is-value* is is-goal × is-goal-list × is-int × is-env.

The denotational continuation semantics of PROLOG follows. Each VDL instruction gives rise to one or more equations whose effect directly mimics the VDL instruction. In the case of a value returning instruction, there is one equation given rise to. If in the VDL instruction, a state component \(C\) is modified to \(C'\), then in the equation, \(C'\) is passed to the continuation as the value of its \(C\) argument. In this respect, passing a value up the control tree is viewed as simply passing that value to the continuation’s value argument. A macro instruction generating \(n\) new nodes to the control tree is transformed into \(n-1\) equations, each doing the next single instruction of the macro and passing its state changes, if any, up to the remaining instructions of the macro expansion and then on to the original continuation.

*Denotational Semantics of PROLOG Derived from Denotational Normal Form VDL Definition:*
Syntax of PROLOG Programs described in VDL:

- `is-program = is-goal-list`
- `is-goal = (is-body-list,is-body-list,is-level,is-n)`
- `is-body = is-term \lor is-cut \lor is-database-op`
- `is-term = is-var \lor is-pred`
- `is-var = \{set of variable identifiers\}`
- `is-pred = (is-pname,is-atom,is-args,is-term-list)`
- `is-atom = \{set of atoms\}`
- `is-cut = \{!\}`
- `is-database-op = (is-name,is-op-name,is-args,is-clause)`
- `is-op-name = \{ASSERT, RETRACT\}`
- `is-clause = (is-head,is-pred,is-body-list,is-body-list)`
- `is-n = \{set of integers greater than equal to 0\}`

Syntax of (run-time) State components described in VDL:

- `is-backtrack-point-stack = (is-c,is-dbptr,is-level,is-btps)`
- `is-dbptr = (is-n,is-pname)`
- `is-database = \{is-clause-list\}`
- `is-output = \{is-term\}`
- `is-env = \{is-pair\}`
- `is-binding = \{is-value\}`
- `is-value = \{is-database,is-backtrack-point-stack\}`
- `is-continuation = \{is-var\}`
- `is-n = \{is-pair\}`
is-pair = (is-term,is-term),is-level,is-n)

Semantic Equations: In the following:

is-n(n)
is-env(env) & is-env(env1)
is-database-pointer(DBPTR) & is-database-pointer(DBPTR1)
is-database(DB) & is-database(DB1)
is-backtrack-point-stack(BTPS) & is-backtrack-point-stack(BTPS1)
is-output(OUTPUT) & is-output(OUTPUT1)
is-c(C) & is-c(C1)
is-value(v)
is-program(p)

The meaning of a program p in a database DB is given as

$M[seek] \ p \ 0 \ \Omega \ DB \ \Omega \ C$

Where: I = the identity continuation.

1. $M[seek] \ goals \ n \ env \ DBPTR \ DB \ BTPS \ OUTPUT \ C =$
   is-<goals> $ightarrow$
   $M[print-bindings] \ env \ DBPTR \ DB \ BTPS \ OUTPUT \ C$
   is-<bodylist1>) $ightarrow$
   $M[seek] \ tail(goals) \ n \ env \ DBPTR \ DB \ BTPS \ OUTPUT \ C$
   is-database-op(body1)) $ightarrow$
   $M[do-database-op] \ body1 \ DBPTR \ DB \ BTPS \ OUTPUT \ C$
   is-var(body1)) $ightarrow$
   $M[seek] \ var-deref \ n \ env \ DBPTR \ DB \ BTPS \ OUTPUT \ C$
   $T \rightarrow$
   $M[set-database-pointer] \ firstgoal \ DBPTR \ DB \ BTPS \ OUTPUT \ C1$

Where:

$C1 \ v \ DBPTR \ DB \ BTPS \ OUTPUT =$

$M[database-seek] \ firstgoal \ restgoals \ n \ env \ DBPTR \ DB \ BTPS \ OUTPUT \ C$

goal1 = elem(1)(goals)
level1 = s-level(goal1)
bodylist1 = s-body-list(goal1)
body1 = elem(1)(bodylist1)
firstgoal = μ0(<s-body-list:body1>,<s-level:level1>)
restgoals = δ(goals;elem(1)s-body-list-elem(1))
var-deref = lookup(μ0(<s-term:body1>,<s-level:level1>))

For: is-goal-list(goals) & is-n(n) & is-env(env)
2. $M[\text{set-database-pointer}]$ goal DBPTR DB BTPS OUTPUT C =
   C $\Omega$ DBPTR1 DB BTPS OUTPUT

Where:
   DBPTR1 = $\mu_0(<s\cdot n:1>, <s\cdot pname>s\cdot pname\cdot elem(1)\cdot (goal)>)$

For: is-goal(goal) & is-pred(elem(1)$\cdot$s-body$\cdot$list(goal)) &
   length(s-body$\cdot$list(goals)) = 1

3. $M[\text{print-bindings}]$ env DBPTR DB BTPS OUTPUT C =
   is-$\Omega$(env) $\rightarrow$
   $M[\text{null}]$ DBPTR DB BTPS OUTPUT C

   T $\rightarrow$
   $M[\text{print-term-and-remove-var}]$ var env DBPTR DB BTPS
   OUTPUT C1

Where:
   C1 env DBPTR DB BTPS OUTPUT =
   $M[\text{print-bindings}]$ env1 DBPTR DB BTPS OUTPUT C
   $\rightarrow$ is-$\Omega$(0$\cdot$var(env))

For: is-env(env)

4. $M[\text{print-term-and-remove-var}]$ var env DBPTR DB BTPS OUTPUT C =
   C $\delta$(env, var) DBPTR DB BTPS OUTPUT1

Where:
   OUTPUT1 = $\mu$(OUTPUT; <var:value(var, 0, env)>)

For: is-var(var) & is-env(env)

5. value(var, n, env) = value-pair(n$\cdot$var(env), env)
   For: is-var(var) & is-n(n) & is-env(env)

6. value-pair(pair, env) =
   is-var(s-term(pair)) $\rightarrow$
   value(s-term(pair), s-level(pair), env)
   is-pred(s-term(pair)) $\rightarrow$
   is-$\langle$(s-args(s-term(pair))) $\rightarrow$
   $\mu_0(<s\cdot p-name>s\cdot p-name\cdot s\cdot term(pair)>)$

   T $\rightarrow$
   $\mu_0(<s\cdot p-name>s\cdot p-name\cdot s\cdot term(pair)>,$
\[ \mu_0(\langle \text{elem}(i).\text{value}(\text{elem}(i)\cdot \text{args}\cdot \text{term}(\text{pair})), \\
\text{s-level}(	ext{pair})\cdot \text{env} \rangle | \]
\[ 1 \leq i \leq \text{length}(\text{s-args}\cdot \text{s-term}(\text{pair})) \rangle ) \]

For: is-pair(pair) & is-env(env)

7. \text{M[database seek]} goal goals n env DBPTR DB BTPS OUTPUT C =
   \text{number-of-clauses} = 0 \rightarrow
   \text{M[fail]} DBPTR DB BTPS OUTPUT C
   \text{number-of-clauses} = 1 \rightarrow
   \text{M[get-clause-head]} n DBPTR DB BTPS OUTPUT C2

T \rightarrow
   \text{M[set-backtrack-point-stack]} n env DBPTR DB BTPS OUTPUT C1

Where:
\[ \text{C1} \vee DBPTR DB BTPS OUTPUT = \]
\[ \text{M[gucs]} \text{pair goals n env DBPTR DB BTPS OUTPUT C} \]
\[ \text{C2} \text{clausehead DBPTR DB BTPS OUTPUT} = \]
\[ \text{M[ucs]} \text{clausehead pair goals n env DBPTR DB BTPS OUTPUT C} \]
\[ \text{number-of-clauses} = \text{length} (\text{s-pname(DBPTR)}(\text{DB})) \]
\[ \text{pair} = \mu_0(\langle \text{s-level} \cdot \text{s-level}(\text{goal}) \rangle, \langle \text{s-term} \cdot \text{elem}(1) \cdot \text{s-body-list}(\text{goal}) \rangle) \]

For: is-goal(goal) & is-goal-list(goals) & is-n(n) & is-env(env)

7a. \text{M[gucs]} pair goals n env DBPTR DB BTPS OUTPUT C =
    \text{M[get-clause-head]} n DBPTR DB BTPS OUTPUT C1

Where:
\[ \text{C1} \text{clausehead DBPTR DB BTPS OUTPUT} = \]
\[ \text{M[ucs]} \text{clausehead pair goals n env DBPTR DB BTPS OUTPUT C} \]

For: is-pair(pair) & is-goal-list(goals) & is-n(n) & is-env(env)

7b. \text{M[ucs]} clausehead pair goals n env DBPTR DB BTPS OUTPUT C =
    \text{M[unify]} clausehead pair env DBPTR DB BTPS OUTPUT C1

Where:
\[ \text{C1 newenv DBPTR DB BTPS OUTPUT} = \]
\[ \text{M[cs]} \text{pair goals newenv n env DBPTR DB BTPS OUTPUT C} \]

For: is-pair(pair) & is-goal-list(goals) & is-n(n) & is-env(env) & is-pred(clausehead)
7c. $M[cs]$ pair goals newenv n env DBPTR DB BTPS OUTPUT C = $M[build-new-goals]$ goals n DBPTR DB BTPS OUTPUT C1

Where:
C1 newgoals DBPTR DB BTPS OUTPUT = $M[seek]$ newgoals n+1 newenv DBPTR DB BTPS OUTPUT C

For: is-pair(pair) & is-goal-list(goals) & is-n(n) & is-env(env) & is-env(newenv)

8. $M[get-clause-head]$ n DBPTR DB BTPS OUTPUT C = $C \mu_0(<s-term:s-head(clause)>, <s-level:n>)$ DBPTR DB BTPS OUTPUT

Where:
clause = $s-n(DBPTR)\langle s-pname(DBPTR)(DB) \rangle$

For: is-n(n)

9. $M[build-new-goals]$ goals n DBPTR DB BTPS OUTPUT C = $C <goal>^goals$ DBPTR DB BTPS OUTPUT

Where:
goal = $\mu_0(<s-body:s-body(clause)>, <s-level:n>)$
clause = $s-n(DBPTR)\langle s-pname(DBPTR)(DB) \rangle$

For: is-goal-list(goals)

10. $M[set-backtrack-point-stack]$ n DBPTR DB BTPS OUTPUT C = $C \Omega DBPTR DB BTPS1 OUTPUT$

where:
BTPS1 = $\mu_0(<s-c:C>,$
<s-dbptr:$\mu_0($
<s-n:$s-n(DBPTR)+1>,$<s-pname:$s-pname(DBPTR)>),
<s-level:n>,
<s-btps:BTPS>)$

11. $M[fail]$ DBPTR DB BTPS OUTPUT C =

is-$\Omega$(clause) →
$C DBPTR DB s-btps(BTPS) C1$

T →
$C DBPTR1 DB BTPS OUTPUT C2$

Where:
\[\text{DBPTR1} = \mu_0(<s\text{-}pname:s\text{-}pname(s\text{-}dbptr(BTPS))>,<s\text{-}n:s\text{-}n\text{-}s\text{-}dbptr(BTPS)>)\]
\[\text{clause} = s\text{-}n(s\text{-}dbptr(BTPS))\text{-}(s\text{-}pname(s\text{-}dbptr(BTPS)))(\text{BoldDB})\]
\[\text{C1} = M[\text{fail}] \text{ DBPTR DB BTPS OUTPUT C}\]
\[\text{C2} = M[s\text{-}c(BTPS)] \text{ DBPTR DB BTPS OUTPUT C}\]

12. \(M[\text{cut\text{-}stack\text{-}to\text{-}level}] n \text{ DBPTR DB BTPS OUTPUT C} =\)
\[s\text{-}level(BTPS) = n\rightarrow\]
\[M[\text{pop\text{-}backtrack\text{-}point\text{-}stack}] n \text{ DBPTR DB BTPS OUTPUT C}\]
\[Tightarrow\]
\[M[\text{pop\text{-}backtrack\text{-}point\text{-}stack}] n \text{ DBPTR DB BTPS OUTPUT C1}\]

Where:
\[C1 \lor \text{ DBPTR DB BTPS OUTPUT} =\]
\[M[\text{cut\text{-}stack\text{-}to\text{-}level}] n \text{ DBPTR DB BTPS OUTPUT}\]

13. \(M[\text{pop\text{-}backtrack\text{-}point\text{-}stack}] \text{ DBPTR DB BTPS OUTPUT C} =\)
\[C \Omega \text{ DBPTS DB s\text{-}btps(BTPS)}\]

14. \(M[\text{unify}] t_1 t_2 \text{ env DBPTR DB BTPS OUTPUT C} =\)
\[\neg\text{is}\text{-}\Omega(\text{lookup}(t_1,\text{env}))\rightarrow\]
\[M[\text{unify}] \text{ lookup}(t_1,\text{env}) t_2\text{env DBPTR DB BTPS OUTPUT C}\]
\[\neg\text{is}\text{-}\Omega(\text{lookup}(t_2,\text{env}))\rightarrow\]
\[M[\text{unify}] t_1 \text{ lookup}(t_2,\text{env}) \text{ env DBPTR DB BTPS OUTPUT C}\]
\[\text{is}\text{-}\text{var}(s\text{-}term(t_1))\rightarrow\]
\[C \mu(\text{env};<s\text{-}level(t_1)s\text{-}term(t_1):t_2>) \text{ DBPTR DB BTPS OUTPUT}\]
\[\text{is}\text{-}\text{var}(s\text{-}term(t_2))\rightarrow\]
\[C \mu(\text{env};<s\text{-}level(t_2)s\text{ term}(t_2):t_1>) \text{ DBPTR DB BTPS OUTPUT}\]
\[\text{functor}(t_1) = \text{functor}(t_2) \& \text{arity}(t_1) = \text{arity}(t_2)\rightarrow\]
\[M[\text{unify\text{-}args} \text{ arity}(t_1) t_1 t_2 \text{ env DBPTR DB BTPS OUTPUT C}\]

For: is\text{-}pair(t_1) \& is\text{-}pair(t_2) \& is\text{\text{-}env(env)}

15. functor(t) = s\text{-}pname(s\text{-}term(t))

For: is\text{-}pair(t)

16. arity(t) = length(s\text{-}args(s\text{-}term(t)))

For: is\text{-}pair(t)

17. lookup(t,\text{env}) = s\text{-}level(t)s\text{-}term(t)(\text{env})
For: is-pair(t) & is-env(env)

18. \[ M[\text{unify-args}] \ r \ t1 \ t2 \ env \ DBPTR \ DB \ BTPS \ OUTPUT \ C = \]
\[ r = 0 \rightarrow \]
\[ C \ env \ DBPTR \ DB \ BTPS \ OUTPUT \]
\[ T \rightarrow \]
\[ M[\text{unify}] \ arg1 \ arg2 \ env \ DBPTR \ DB \ BTPS \ OUTPUT \ C1 \]

Where:
\[ C1 \ newenv \ DBPTR \ DB \ BTPS \ OUTPUT = \]
\[ M[\text{unify-args}] \ r-1 \ t1 \ t2 \ newenv \ DBPTR \ DB \ BTPS \ OUTPUT \ C \]
\[ \text{arg1} = \mu_{0}(<s-term:elem(r)(s-args(s-term(t1))), <s-level:s-level(t1)> ) \]
\[ \text{arg2} = \mu_{0}(<s-term:elem(r)(s-args(s-term(t2))), <s-level:s-level(t2)> ) \]

For: is-int(r) & is-pair(t1) & is-pair(t2) & is-env(env)

19. \[ M[\text{do-database-op}] \ dbop \ DBPTR \ DB \ BTPS \ OUTPUT \ C = \]
\[ s-name(dbop) = ASSERT \rightarrow \]
\[ C \ \Omega \ DBPTR \ \mu(DB;\langle p-name:clause-list+\rangle) \ BTPS \ OUTPUT \]
\[ s-name(dbop) = RETRACT \rightarrow \]
\[ C \ \Omega \ DBPTR \ \mu(DB;\langle p-name:clause-list-\rangle) \ BTPS \ OUTPUT \]

Where:
\[ \text{clause-list} = p-name(DB)^{-}\langle \text{clause} \rangle \]
\[ \text{clause-list}- = \text{delete}(p-name(DB), \text{clause}) \]
\[ \text{delete}(l,e) = (\text{is-list}(l) \rightarrow \]
\[ (\neg \exists \ i)(\ 1 \leq i \leq \text{length}(l) \rightarrow \text{elem}(i,l) = e) \rightarrow \]
\[ T \rightarrow (\text{elem}(i,l) = e \rightarrow \]
\[ \mu_{0}((\{<\text{elem}(j,l)\mid 1 \leq j \leq i-1\})^{*} \]
\[ \mu_{0}((\{<\text{elem}(j,l)\mid i+1 \leq \text{length}(l)\})^{*})) \]
\[ p-name = s-pname(s-head(clause)) \]
\[ \text{clause} = s-args(dbop) \]

For: is-database-op(dbop)

4. CONCLUSION

This paper has given two formal semantics for PROLOG, one operational and one denotational. These semantics cover the full language including the database and extralogical features. In addition, the operational semantics assigns a non-degenerate meaning to a program written as a perpetual process, i.e., a program that never halts but is doing useful work. The meaning of such a program is a infinite sequence of states exhibiting the useful work done.
In this paper, the semantics of PROLOG has been given in two different styles, VDL and denotational continuation. The first of these was based on a cited PROLOG Horn clause definition. Then the second was obtained by systematic transformation from the first. It is also possible to systematically transform the denotational, continuation definition into a purely synthesized attribute grammar [4] definition and then to systematically transform this attribute grammar into PROLOG [2], thus completing the cycle.

After having done the construction of the denotational semantics by the method of [3], it is easy in retrospect to see how to define the database and the extra-logical feature, and for that matter any feature, denotationally. One simply lets the meaning of the feature be the application of the current continuation to the components of the state that get modified by the feature.

The authors thank Yossi Betser, Peter Lucas, and the referees for their helpful comments.

REFERENCES

14. Parker, D. S. and Eggert, P. R., A Prolog Interpreter, Private Communication.