TECHNICAL REPORT NO. 292

A Prolog Semantics with First-Class Continuations and Dynamic Database

by

Christopher T. Haynes, Indiana University

and

Richard M. Salter, Oberlin College

Revised: October 1990

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

A Prolog Semantics with First-Class Continuations and Dynamic Database*

(Revised October, 1990)

Christopher T. Haynes
Indiana University†

Richard M. Salter
Oberlin College‡

Keywords

Logic programming, control, continuations, dynamic binding, denotational semantics, Prolog, cut

Abstract

The depth-first control behavior of most logic programming languages, such as Prolog, is powerful but inflexible. The language Conlog extends Prolog with extra-logical operators that allow alternate control behavior to be expressed, while retaining a default depth-first behavior. In this paper we describe Conlog's primitive control operators, which allow the programmer to bind the current success or failure continuation to a variable and invoke it at any future time. These continuations are first-class objects that may be stored in the database. Together with an operator for dynamically restoring database bindings, this allows a variety of control behavior, including variations on breadth-first search and multitasking, to be conveniently expressed. A denotational semantics of Conlog is presented that accounts for the intensional nature of continuations, as opposed to the extensional character of other Prolog data. Cut is also expressed in a unique way, and a careful treatment of the traditional assert and retract operations is included.

* This research was supported by NSF grants CCR-8702117 and CCR90-00597.

† Computer Science Dept., Bloomington, IN 47405 USA; chaynes@cs.indiana.edu.

‡ Computer Science Program, Oberlin, OH 44074 USA; rms@occs.oberlin.edu.

1. Introduction

Logic programming, as exemplified by Prolog, implements deductive inference on Horn clauses through an efficient but inflexible depth-first control structure. The extra-logical cut operator provides limited ability to prune the search tree, but does not alter the depth-first character of the search. Other control strategies, such as variations on breadth-first search, can be achieved using interpretation (meta-programming) or self-modifying programs, but this generally results in inefficient and obscure programs.

Meta-rules have also been proposed to control clause and literal selection, and these may be used to implement breadth-first search. [9] However, this technique is based on a fixed set of properties reflecting the state of the interpreter. As a result, it is not possible to use data-dependent heuristics, as in Algorithm A. [16] The restriction to a fixed set of properties for meta-control is critical. If control and logic were not divorced in this way, constant reinterpretation of the meta-rules would be required.

The language Conlog is an extension of Prolog that includes several novel extra-logical operators. These operators allow alternate control strategies to be directly expressed within the framework of Prolog. The programmer benefits from the efficiency and power of a default depth-first search control regime, while at the same time retaining the ability to obtain radically different control behavior as needed.

In this paper we describe the primitive control operators of Conlog. These operators provide the ability to capture, or *reify,* the control context at any point in a program and return to this context at any future time. The captured control information is encapsulated as a *continuation* that may be manipulated as *first-class* data with unlimited extent. Continuations may thus be bound to logic variables and stored in the database.

Continuations abstract the control context in which the "future" of computations would normally be played out. Languages that support backtracking are readily modeled using two continuation types: *success continuations* that continue forward progress and *failure continuations* that perform backtracking. In logic programming, the control context includes the choice points used when backtracking and, for failure continuations, the current bindings of logic variables.

In some applications the limited scope of logic variables requires that continuations be stored in the database. Unfortunately, the traditional Prolog database is global (unscoped), which may makes it difficult or impossible to maintain the appropriate environment for computations involving continuations. This has motivated a Conlog operator that provides dynamic database protection. This operator automatically restores database bindings of a given relation, R, when control leaves a given goal, G. Since any assertions to R by G have dynamic extent, they may be viewed as belonging to a "hypothetical world" constructed by G.

Though first-class continuations are a general abstraction of control, their low level makes it difficult to directly abstract common paradigms of control behavior. In a companion paper we present the high-level control operators of Conlog, which allow control patterns such as breadth-first search and multi-tasking (coroutining) to be conveniently expressed. [20] These high-level operators may be implemented as primitives, or they may be programmed using the low-level operators introduced in this paper.

A denotational semantics of Conlog's primitive control operators is presented in this paper. This semantics is based on earlier work, notably that of Jones and Mycroft [13] and de Bruin and de Vink [4], but is unique in its treatment of the operators introduced here. Of particular interest is the intensional treatment of continuations as data, which contrasts with the extensional nature of other data and influences the semantics at several points. The semantics also makes use of the dynamic database to express cut in a novel way and provides a careful treatment of the traditional assert and retract operations.

The next section presents the Conlog operators for accessing and invoking first-class continuations. Section 3 describes the database protection operator. Section 4 contains the denotational specification of Conlog. Section 5 contains concluding remarks. Familiarity with Prolog, first-class continuations, and the techniques of denotational semantics is assumed. [6,11,21]

2. First-class continuations in logic programming

In this section we introduce the extra-logical operators `csc`, `cfc`, `cut_to`, and `throw`. The `csc` and `cfc` operators reify success and failure continuations, respectively, while the `cut_to` and `throw` operators are used to invoke reified success and failure continuations, respectively.

The operator `cfc` is called with an unbound logic variable and returns with the variable bound to the current failure continuation, represented as a first-class object. Reified continuations have indefinite extent and may be asserted into the database as fact arguments.

The operator `cut_to` takes a single argument, which must be a variable bound to a reified continuation object. The continuation represented by the object replaces the current failure continuation. This is analogous to the behavior of cut, which replaces the current failure continuation with the failure continuation of the current relation's entry point. If backtracking causes the new failure continuation to be invoked, it will continue from the point at which the continuation was created. When the continuation is invoked, logic variable and protected database bindings are restored to the values current at the time the continuation was created.

The following program illustrates the use of `cfc` and `cut_to`.

```
p(X,Y) :- X = 3, cfc(FC), F =.. [fail_cont, FC], asserta(F), q(Y).
p(X,Y) :- X = 4.
q(Y) :- Y = 5, retract(fail_cont(FC1)), cut_to(FC1), fail.
```

Invoking `p` first binds `X` to three, and then binds `FC` to the current failure continuation, which begins by unbinding `X`. Next a structure is constructed, with the `=..` operator, that has functor `fail_cont` and the reified failure continuation as its argument. This fact is then asserted into the database. The call "`q(Y)`" first binds Y to five and then binds the reified continuation to FC1, by matching the relation fail_cont in the database. The fail_cont relation is then retracted, making further access to the continuation impossible. Finally, FC1 is invoked by first using cut_to to substitute it for the current failure continuation and then executing fail to initiate backtracking. This returns control to the first clause of p at the point following the binding of X to three (so that Y is no longer bound to five), and proceeds by revoking the binding of X to three. The computation then enters the second clause of p.

The csc operator takes an unbound logic variable and a subgoal. It binds the variable to the current success continuation (represented as a first-class object), which is the success continuation of the csc call, and then calls the given subgoal. If the subgoal succeeds, the success continuation is implicitly invoked.

Finally, the throw operator takes a logic variable, which must be bound to a reified success continuation, and invokes this continuation. This effectively forces success of the subgoal associated with the csc statement that created the continuation. Logic variable bindings and choice points established by the subgoal are retained. Thus subsequent failure will back control into the subgoal. However, a subsequent cut in the rule containing the success continuation's csc statement will cut the choice points of the csc subgoal as well as those of the rule.

For example, consider the following program.

```
p(X) :- csc(SC, q(SC,X)), fail.
p(X) :- csc(SC, q(SC,X)), !.
q(SC,Y) :- Y = 1, throw(SC).
q(SC,2).
```

Invoking p first binds SC to a reified success continuation and then invokes the subgoal q(SC,X). The subgoal returns with X bound to 1, but then fail sends control back into q. The first clause of q then fails and the second clause binds X to 2. Fail then sends control back into q, which now fails, causing the first clause of p to fail. The second clause of p then calls q again. This time p succeeds with X bound to 1 when control returns, but the choice points of p and q are cut first. Thus if control fails back into p, it will fail without trying the second clause of q.

In Prolog, user defined procedures have an extensional representation that may be manipulated and printed. However, in most procedural languages they do not. This is for reasons of abstraction and efficiency, and is in keeping with the usual intensional treatment of functions in mathematics. For similar reasons continuations should not have an extensional representation, in either traditional languages or in the context of logic programming. The intensional nature of continuations means that they have no external representation; they cannot be printed as results or decomposed. Furthermore, equality of such intensional objects is undecidable. Thus to maintain the abstract nature of continuations we feel that unification should not be used to determine if two continuation objects are the same. Consequently we add the restriction that reified success and failure continuations unify only with unbound variables.

3. Dynamic database

Logic variables do not always suffice for recording continuations. The automatic restoration of logic variable bindings, when failure continuations are invoked, may prohibits the use of logic variables for communication between independent program threads. Thus the database must at times be used to communicate continuations. (Stylistically the database may also be preferable, even when it is possible to communicate through logic variables, for its use may reduce the need for extensive parameter passing.) Unfortunately, the global character of the database is problematic. In addition to the possibility of accidental naming conflicts, there is a problem when a procedure that makes database assertions is called recursively. Care must be taken not to confuse assertions made at different levels of recursion.

In the absence of continuations that outlive their dynamic context, the latter problem may be handled by recording a database binding in a local variable and then retracting the database binding upon entry to a recursive procedure. The old binding may then be re-asserted before the procedure returns. For example, in the following program schema procedure `p` may use the argument of `foo`'s database binding to communicate information to procedures that it calls.

```
p :- retract(foo(Save)), ..., asserta(foo(x)), ..., p, ...,
     retract(foo(New)), F =.. [foo,Save], asserta(F).
```

When called, `p` saves the current argument of the initial clause of `foo` in the variable `Save`, retracts the old clause, and then asserts a new clause before calling itself recursively. Before `p` returns to its caller, the process is reversed to restore the old clause of `foo`.

This technique is not reliable in the presence of first-class (success or failure) continuations. Control may pass out of the dynamic context of a procedure through invocation of continuations without restoring database bindings. The same problem arises in other programming languages, such as Scheme [18], that support first-class continuations. The solution is to provide a *dynamic environment* whose bindings are restored when a continuation is invoked to their values at the time the continuation was reified. Rather than providing a separate dynamic environment in Conlog, we achieve the same effect with the `protect` operator, which dynamically protects database bindings. Protected database bindings may be viewed as entries in a dynamic database environment that shadows the global database. (This is the technique by which database protection is modeled in the semantics of Section 4.)

If R is a relation name and G is a goal, then a call of the form `protect` (R, G) saves the current database binding of R and calls G. The saved binding of R is restored whenever control leaves G, either through success, backtracking, or continuation invocation. If control returns to G through backtracking or continuation invocation, the binding is returned to its value at the time control left the context. Thus the rest of the program is protected from manipulation of relation R by G, and G is protected from manipulation of relation R by the rest of the program.

Dynamic database bindings may be used to implement cut for a given procedure `p` as follows. First `p` is renamed within its own definition, say to `p1`. Then `p` is redefined to be an auxiliary procedure that uses the procedure `cut_call` to invoke `p1`. Within `p1` the cut operation is performed by calling the procedure `cut`, which retrieves the current cut continuation from the database, where it was stored by `cut_call`, and installs it with `cut_to`.

```
    p(t1,...,tn) :- cut_call(p1(t1,...,tn)).
    cut_call(G) :- cfc(CutFC),
                   protect(current_cut/1, (retract(current_cut(X)),
                                           S =.. [current_cut, CutFC],
                                           asserta(S), call(G))).
    cut :- current_cut(CutFC), cut_to(CutFC).
```

`cut_call` first reifies the failure continuation of the forthcoming call of `G`. The relation `current_cut` (with arity 1) is then protected. Within the scope of this protection any existing binding of `current_cut` is retracted and a new binding is asserted that includes the reified continuation, after which the goal `G` is called. The retracted binding of `current_cut` is restored by the protection mechanism when control leaves goal `G` with success or failure.

4. Denotational semantics of Conlog

In this section we present a denotational semantics of Prolog, extended to include the `cfc`, `cut_to`, and `protect` operations. This semantics also treats cut in a novel way and is careful in its treatment of assert and retract.

An abstract syntax for Conlog is given in Figure 1. For every variable V in the (countably) infinite set of variables *Var*, *Var* is assumed to contain, for each natural number n, a distinct variable denoted $V^{(n)}$ and referred to as the *renaming of V at level n*. This domain allows an already renamed variable, say $V^{(n)}$, to be renamed again. Each renaming yields a distinct variable, so for example $V^{(n_1)(m)}$ and $V^{(n_2)(m)}$ are distinct. Variables may be represented by a base variable (one that has not been renamed) with an associated list of renaming levels.

A new renaming level is used for each access to the database, and asserted clauses may include variables. As a result, the present semantics requires the ability to rename already renamed variables. For example, when the procedure

```
    p :- asserta(a(X)), a(Y).
```

is called, it is first retrieved from the database, at which time `X` is renamed, say to $\mathtt{X}^{(n)}$. $\mathtt{a(X}^{(n)}\mathtt{)}$ is then asserted as a fact. When this fact is retrieved by the goal `a(Y)`, $\mathtt{X}^{(n)}$ is renamed again, say to $\mathtt{X}^{(n)(m)}$, before being bound to $\mathtt{Y}^{(n)}$. In a semantics without database assertion, only base variables need be renamed. [4,13]

Relation identifiers include the cut (`!`) and list construction (`.`) operators. It is presumed that arity information is encoded in relation identifiers in the translation from concrete to abstract syntax. Constants include the empty list (`[]`). Facts are treated as clauses with the body `true`. *Term** and *Clause** indicate sequences of zero or more terms and clauses, respectively. An empty term sequence is associated with unparameterized relation identifiers, including cut.

$$
\begin{array}{rrl}
 & \mathit{Keyword} ::= & \texttt{true} \mid \texttt{fail} \mid \texttt{and} \mid \texttt{or} \mid \texttt{call} \mid \texttt{cfc} \mid \texttt{cut\_to} \mid \texttt{protect} \\
 & & \mid \texttt{csc} \mid \texttt{throw} \mid \texttt{asserta} \mid \texttt{assertz} \mid \texttt{retract} \mid \leftarrow \mid = \\
V \in & \mathit{Var} & \text{Variable identifiers (includes } V^{(n)} \text{ for all } V \in \mathit{Var}) \\
R \in & \mathit{Relid} & \text{Relation identifiers (includes } \texttt{!} \text{ and } \texttt{.}\,) \\
F \in & \mathit{Functor} ::= & \mathit{Keyword} \mid \mathit{Relid} \\
K \in & \mathit{Const} ::= & \mathit{Var} \mid \mathit{Functor} \mid \texttt{[]} \mid \ldots \\
S \in & \mathit{Structure} ::= & \mathit{Functor}(\mathit{Term}^*) \\
T \in & \mathit{Term} ::= & \mathit{Const} \mid \mathit{Structure} \\
G \in & \mathit{Goal} ::= & \texttt{true} \mid \texttt{fail} \mid \mathit{Relid}(\mathit{Term}^*) \mid \mathit{Term}_1 = \mathit{Term}_2 \mid \texttt{call}\,(\mathit{Var}) \\
 & & \mid \mathit{Goal}_1 \texttt{ and } \mathit{Goal}_2 \mid \mathit{Goal}_1 \texttt{ or } \mathit{Goal}_2 \mid \texttt{retract}\,(\mathit{Term}) \\
 & & \mid \texttt{asserta}\,(\mathit{Term}) \mid \texttt{assertz}\,(\mathit{Term}) \mid \texttt{protect}\,(\mathit{Term},\ \mathit{Goal}) \\
 & & \mid \texttt{cfc}\,(\mathit{Var}) \mid \texttt{cut\_to}\,(\mathit{Var}) \mid \texttt{csc}\,(\mathit{Var},\ \mathit{Goal}) \mid \texttt{throw}\,(\mathit{Var}) \\
C \in & \mathit{Clause} ::= & \mathit{Relid}(\mathit{Term}^*) \leftarrow \mathit{Goal} \\
 & \mathit{Program} ::= & \mathit{Clause}^*;\ \mathit{Goal}
\end{array}
$$

Figure 1. Abstract syntax

The domain equations are given in Figure 2. $D_\perp$ denotes domain D lifted by addition of a bottom element. D^* indicates the domain of finite sequences of elements from domain D. $\times$ and $+$ denote smashed product and coalesced sum, respectively. The domain Den of denotable values is a generalization of *Term* that allows success and failure continuations to be recorded in substitutions and databases.

A relation is represented as a sequence of 4-tuples, one for each clause of the relation. Each 4-tuple contains the sequence of values representing the terms of the clause's head, the value representing the clause's body, the denotation of the clause's body, and an index which, if non-zero, identifies a deleted clause. The value representing the body is used only by `retract` .

Though substitutions are abstractly partial functions, we represent them as total functions by extending their ranges to be disjoint unions including domains containing the unique element *unbound*. Natural numbers are used to record variable renaming levels, as in structure sharing implementations [3,4], and as unique indices for the global database. The next unused number is passed to failure continuations. The first renaming level passed to goal denotations is the level at which the goal was accessed in the database.

Dynamically bound denotable values are recorded in the global database, where they are indexed by numbers. If a relation identifier is dynamically bound, the dynamic database maps it to the associated index number. Relation identifiers that are not dynamically bound are fixpoints of the associated dynamic database (modulo injection into the Did domain).

The Dcont domain is that of specialized continuations used by the auxiliary function *delete*. Answers are lists, constructed by $\times$, of answer substitutions. These lists may be either finite (terminated by *fail*) or infinite, in the manner of Jones and Mycroft [13]. Answer substitutions normally map variables to terms or to *unbound*,

$$\begin{array}{rrcll}
m,n \in & \mathrm{N} & = & \{0,1,2,\ldots\}_\perp & \text{Natural numbers} \\
 & \mathrm{Struct} & = & \mathit{Functor}_\perp \times \mathrm{Den}^* & \text{Structured values} \\
\delta \in & \mathrm{Den} & = & \mathit{Const}_\perp + \mathrm{Struct} + \mathrm{Fcont} + \mathrm{Scont} & \text{Denotable values} \\
\rho \in & \mathrm{Rel} & = & (\mathrm{Den}^* \times \mathrm{Den} \times \mathrm{Gden} \times \mathrm{N})^* & \text{Relations} \\
\sigma \in & \mathrm{Subst} & = & \mathit{Var} \to (\mathrm{Den} + \{\mathit{unbound}\}_\perp) & \text{Substitutions} \\
\iota \in & \mathrm{Did} & = & \mathit{Relid}_\perp + \mathrm{N} & \text{Database indices} \\
\beta \in & \mathrm{Ddb} & = & \mathit{Relid} \to \mathrm{Did} & \text{Dynamic databases} \\
\gamma \in & \mathrm{Gdb} & = & \mathrm{Did} \to \mathrm{Rel} & \text{Global databases} \\
\phi \in & \mathrm{Fcont} & = & \mathrm{Gdb} \to \mathrm{N} \to \mathrm{Ans} & \text{Failure continuations} \\
\xi \in & \mathrm{Scont} & = & \mathrm{Fcont} \to \mathrm{Subst} \to \mathrm{Ddb} \to \mathrm{Fcont} & \text{Success continuations} \\
\pi \in & \mathrm{Gden} & = & \mathrm{N} \to \mathrm{Scont} \to \mathrm{Scont} & \text{Goal denotations} \\
\kappa \in & \mathrm{Dcont} & = & \mathrm{Rel} \to \mathrm{Subst} \to \mathrm{N} \to \mathrm{Ans} & \textit{delete} \text{ continuations} \\
 & \mathrm{Ans} & = & (\mathrm{Asubst} \times \mathrm{Ans}_\perp) + \{\mathit{fail}\}_\perp & \text{Answers} \\
 & \mathrm{Asubst} & = & \mathit{Var} \to (\mathit{Term}_\perp + \{\mathit{fail}, \mathit{unbound}\}_\perp) & \text{Answer substitutions}
\end{array}$$

Figure 2. Domain equations

which indicates that the substitution does not associate a value with the given variable. However, answer substitutions may also return *fail*, indicating the value associated with the given variable contains a failure continuation and thus cannot be represented as a term.

The valuation functions are given in Figure 3. Braces are used for grouping, while parentheses and angle brackets are used to indicate product and sequence construction, respectively. $\vec{x}$ indicates a sequence of elements from domain x. § indicates concatenation of the sequences, while $x : \rho$ abbreviates $\langle x \rangle \ \S\ \rho$. $\vec{x}_{(i)}$ represents the element of sequence $\vec{x}$ with index i.

The program valuation $\mathcal{P}$ calls the goal valuation $\mathcal{G}$ with a global database into which the clauses have been loaded by the clause valuation $\mathcal{C}$. All global databases are formed by functional extension of this database. The initial dynamic database β_0 is the identity function on relation identifiers. If a global database does not bind a given relation, then γ_0 associates it with an empty relation sequence. This causes predicates referring to the relation to fail. The initial failure continuation ϕ_0 returns *fail*, terminating an answer list. The initial success continuation ξ_0 constructs an answer list headed by an answer substitution and followed by the answer list obtained by invoking the failure continuation ϕ. The auxiliary function *ans* (see Figure 4) converts the substitution σ passed to ξ_0 into an answer substitution. Numerous domain injections and projections, such as the injection of $\langle\rangle$ into Rel and *fail* into Ans, have been omitted for clarity.

The clause valuation loads program clauses into the global database. Each clause of the form $R(\vec{T}) \leftarrow G$ is represented as a triple consisting of the sequence of values representing $\vec{T}$, the value representing G, and the denotation of G. The (postfix) auxiliary function $\downarrow$ performs *down conversion,* returning the value associated with a term, while $\Downarrow$ returns a sequence of values obtained by down converting a sequence of terms. The triple is placed at the end of a sequence denoting those

$\mathcal{P} : \mathit{Program} \rightarrow \mathrm{Ans}$

$\mathcal{P}[\![\vec{C};\ G]\!] = \mathcal{G}[\![G]\!]0\xi_0\phi_0\sigma_0\beta_0\{\mathcal{C}[\![\vec{C}]\!]\gamma_0\}1$

where $\xi_0 = \lambda\phi\sigma\beta\gamma n.(\mathit{ans}\,\sigma, \phi\gamma n)$, $\phi_0 = \lambda\gamma n.\mathit{fail}$, $\sigma_0 = \lambda V.\mathit{unbound}$, $\beta_0 = \lambda R.R$, and $\gamma_0 = \lambda\iota.\langle\rangle$

$\mathcal{C} : \mathit{Clause}^* \rightarrow \mathrm{Gdb} \rightarrow \mathrm{Gdb}$

$\mathcal{C}[\![\langle\rangle]\!]\gamma = \gamma$

$\mathcal{C}[\![R(\vec{T}) \leftarrow G : \vec{C}]\!]\gamma = \mathcal{C}[\![\vec{C}]\!]\gamma'$, where $\gamma' = \gamma[R \mapsto \gamma[\![R]\!]\ \S\ \langle(\vec{T}\Downarrow, G\downarrow, \mathcal{G}[\![G]\!], 0)\rangle]$

$\mathcal{G} : \mathit{Goal} \rightarrow \mathrm{Gden}$

$\mathcal{G}[\![\mathtt{true}]\!]m\xi = \xi$

$\mathcal{G}[\![\mathtt{fail}]\!]m\xi\phi\sigma\beta = \phi$

$\mathcal{G}[\![R(\vec{T})]\!]m\xi\phi\sigma\beta\gamma = \mathit{alt}\ \rho\vec{\delta}m\xi\phi\sigma\beta\gamma$, where $\rho = \gamma\{\beta[\![R]\!]\}$ and $\vec{\delta} = \vec{T}\Downarrow$

$$\mathcal{G}[\![T_1 = T_2]\!]m\xi\phi\sigma\beta = \begin{cases} \xi\phi\{\sigma\sigma'\}\beta, & \text{if } \sigma' = \mathit{mgu}(T_1\downarrow^{(m)}\sigma, T_2\downarrow^{(m)}\sigma) \neq \mathit{fail}; \\ \phi, & \text{otherwise.} \end{cases}$$

$\mathcal{G}[\![G_1\ \mathtt{and}\ \ G_2]\!]m\xi = \mathcal{G}[\![G_1]\!]m\xi'$, where $\xi' = \mathcal{G}[\![G_2]\!]m\xi$

$\mathcal{G}[\![G_1\ \mathtt{or}\ \ G_2]\!]m\xi\phi\sigma\beta = \mathcal{G}[\![G_1]\!]m\xi\phi'\sigma\beta$, where $\phi' = \mathcal{G}[\![G_2]\!]m\xi\phi\sigma\beta$

$$\mathcal{G}[\![\mathtt{call}\,(V)]\!]m\xi\phi\sigma\beta = \begin{cases} \mathcal{G}[\![G]\!]m\xi\phi\sigma\beta, & \text{if } \sigma V^{(m)} \neq \mathit{unbound}; \\ & \text{and } G = \mathit{goal}(\sigma V^{(m)}\uparrow) \neq \mathit{fail}; \\ \phi, & \text{otherwise.} \end{cases}$$

(continued)

Figure 3. Valuation functions

clauses of the relation R that preceded the current clause in the program's clause list.

The goal `true` causes the current success continuation to be invoked. ($\mathcal{G}[\![\mathtt{true}]\!]m\xi = \xi$ is equivalent to $\mathcal{G}[\![\mathtt{true}]\!]m\xi\phi\sigma\beta\gamma n = \xi\phi\sigma\beta\gamma n$.) Similarly, the goal `fail` results in invocation of the current failure continuation.

In response to predications of the form $R(\vec{T})$, the alternation auxiliary function *alt* is invoked with the result of looking up the relation denoted by R. *alt* fails given an empty relation. Otherwise, it attempts to unify the value sequence $\vec{\delta}$ associated with $\vec{T}$ and the value sequence $\vec{\delta}_1$ representing the head of the first clause of the relation. Before unification $\vec{\delta}_1$ is renamed at a new level n and $\vec{\delta}$ is renamed at the level m of the predication and any variables bound by the current substitution σ are replaced by the values indicated by σ. If unification succeeds, returning substitution σ', the denotation π of the clause's body is invoked. π is passed the new indexing level n, a modified success continuation ξ' (discussed below), a failure continuation ϕ' that will attempt to unify $\vec{\delta}$ with other clauses of the relation, a

$$\mathcal{G}[\![\texttt{asserta}\,(T)]\!]m\xi\phi\sigma\beta\gamma = \begin{cases} \xi\phi\sigma\beta\gamma', & \text{if } (R, \vec{\delta}, \delta') = \mathit{unparse}\ T\sigma m \\ & \text{and } G = \mathit{goal}\{\delta'\uparrow\} \neq \mathit{fail}; \\ \phi\gamma, & \text{otherwise.} \end{cases}$$

where $\iota = \beta[\![R]\!]$ and $\gamma' = \gamma[\iota \mapsto (\vec{\delta}, G\downarrow, \mathcal{G}[\![G]\!], 0) : \gamma\iota]$

$$\mathcal{G}[\![\texttt{assertz}\,(T)]\!]m\xi\phi\sigma\beta\gamma = \begin{cases} \xi\phi\sigma\beta\gamma', & \text{if } (R, \vec{\delta}, \delta') = \mathit{unparse}\ T\sigma m \\ & \text{and } G = \mathit{goal}\{\delta'\uparrow\} \neq \mathit{fail}; \\ \phi\gamma, & \text{otherwise.} \end{cases}$$

where $\iota = \beta[\![R]\!]$ and $\gamma' = \gamma[\iota \mapsto \gamma\iota \,\S\, \langle(\vec{\delta}, G\downarrow, \mathcal{G}[\![G]\!], 0)\rangle]$

$$\mathcal{G}[\![\texttt{retract}\,(T)]\!]m\xi\phi\sigma\beta\gamma = \begin{cases} \mathit{delete}\,\{\gamma\iota\}\langle\rangle\vec{\delta}\{\phi\gamma\}\kappa, & \text{if } (R, \vec{\delta}_1, \delta') = \mathit{unparse}\ T\sigma m; \\ \phi\gamma, & \text{otherwise.} \end{cases}$$

where $\iota = \beta[\![R]\!]$, $\vec{\delta} = \delta' : \vec{\delta}_1$, $\kappa = \mathit{fix}\, f.\lambda\rho\sigma' n.\xi\phi'\{\sigma\sigma'\}\beta\{\gamma[\iota \mapsto \rho]\}n$, and $\phi' = \mathit{fix}\, g.\lambda\gamma n'.\mathit{delete}\,\{\mathit{find}\{\gamma\iota\}\{n-1\}\}\langle\rangle\vec{\delta}\{g\gamma\}fn'$

$$\mathcal{G}[\![\texttt{protect}\,(T, G)]\!]m\xi\phi\sigma\beta\gamma n = \begin{cases} \mathcal{G}[\![G]\!]m\xi'\phi\sigma\beta'\gamma'\{n+1\}, & \text{if } P \text{ or } Q; \\ \phi\gamma n, & \text{otherwise.} \end{cases}$$

where $\delta = \sigma\{T \mid \mathrm{Var}\}^{(m)}$, $\xi' = \lambda\phi\sigma\beta_1.\xi\phi\sigma\beta$, $\beta' = \beta[R \mapsto n]$, $\gamma' = \gamma[n \mapsto \gamma\{\beta[\![R]\!]\}]$, $P =$ "$T \mathrel{\mathsf{E}} \mathit{Var}$, $\delta \neq \mathit{unbound}$, $\delta \mathrel{\mathsf{E}} \mathit{Relid}$, and $R = \delta \mid \mathit{Relid}$," and $Q =$ "$T \mathrel{\mathsf{E}} \mathit{Relid}$ and $R = T \mid \mathit{Relid}$."

$$\mathcal{G}[\![\texttt{cfc}\,(V)]\!]m\xi\phi\sigma\beta = \begin{cases} \xi\phi\{\sigma\sigma'\}\beta, & \text{if } \sigma' = \mathit{mgu}(\langle V{\downarrow}^{(m)}\sigma\rangle, \langle\phi \text{ in Den}\rangle) \neq \mathit{fail}; \\ \phi, & \text{otherwise.} \end{cases}$$

$$\mathcal{G}[\![\texttt{cut\_to}\,(V)]\!]m\xi\phi\sigma\beta = \begin{cases} \xi\{\sigma V^{(m)} \mid \mathrm{Fcont}\}\sigma\beta, & \text{if } \sigma V^{(m)} \mathrel{\mathsf{E}} \mathrm{Fcont}; \\ \phi, & \text{otherwise.} \end{cases}$$

$$\mathcal{G}[\![\texttt{csc}\,(V, G)]\!]m\xi\phi\sigma\beta = \begin{cases} \mathcal{G}[\![G]\!]m\xi\phi\{\sigma\sigma'\}\beta, & \text{if } \sigma' = \mathit{mgu}(\langle V{\downarrow}^{(m)}\sigma\rangle, \langle\delta\rangle) \neq \mathit{fail}; \\ \phi, & \text{otherwise.} \end{cases}$$

where $\delta = \xi'$ in Den and $\xi' = \lambda\phi\sigma\beta_1.\xi\phi\sigma\beta$

$$\mathcal{G}[\![\texttt{throw}\,(V)]\!]m\xi\phi\sigma\beta = \begin{cases} \{\sigma V^{(m)} \mid \mathrm{Scont}\}\phi\sigma\beta, & \text{if } \sigma V^{(m)} \mathrel{\mathsf{E}} \mathrm{Scont}; \\ \phi, & \text{otherwise.} \end{cases}$$

Figure 3. Valuation functions (continued)

substitution obtained by composing σ with σ', a dynamic database β' extended for cut (discussed below), the current global database γ, and the next unused indexing level $n+1$. If unification fails, it is only necessary to invoke ϕ'.

The success continuation ξ' with which a clause is invoked is identical to the success continuation ξ of the call except in one respect: the current dynamic database is replaced by the dynamic database that was current when the clause was entered. Thus the success continuation of a clause is closed over the dynamic database, while the success continuation of a conjunctive goal is not closed over the dynamic database. This is necessary to support our implementation of cut.

The standard way to support cut in a continuation semantics of Prolog is to pass

$mgu : (\text{Den}^* \times \text{Den}^*) \rightarrow (\text{Subst} + \{\mathit{fail}\}_\perp)$

Most general unifier. Success and failure continuations unify only with unbound variables. Definition omitted.

$goal : (\mathit{Term} + \{\mathit{fail}\}_\perp) \rightarrow (\mathit{Goal} + \{\mathit{fail}\}_\perp)$

Natural embedding of *Term* into *Goal*. *goal* preserves the token *fail*, and returns *fail* if the given term is not the prefix form of some goal. Definition omitted.

$\_^{(n)} : (\text{Den} \times \text{N}) \rightarrow \text{Den}$ and $(\text{Den}^* \times \text{N}) \rightarrow \text{Den}^*$

Rename, at level n, all variables (both previously tagged and untagged) in a value or sequence of values. Definition omitted.

$\downarrow : \mathit{Term} \rightarrow \text{Den} \qquad \uparrow : \text{Den} \rightarrow (\mathit{Term} + \{\mathit{fail}\}_\perp)$

$K\downarrow = K \qquad K\uparrow = K,\ \phi\uparrow = \mathit{fail}$

$$F(\vec{T})\downarrow = (F, \vec{T}\Downarrow) \quad (F, \vec{\delta})\uparrow = \begin{cases} \mathit{fail}, & \text{if there exists an } i \text{ such that } \vec{\delta}_{(i)}\uparrow = \mathit{fail}; \\ F(\vec{\delta}\Uparrow), & \text{otherwise.} \end{cases}$$

(continued)

Figure 4. Auxiliary functions

each goal denotation a cut continuation, as well as success and failure continuations. [2,4,7,8,15,17,22] This is avoided in our semantics by dynamically binding the cut symbol, `!`, to a specially created relation when invoking a goal denotation. This relation has a single triple containing an empty sequence and a value representing `true` (as if cut were a fact with arity zero) and a special function with the type of a goal denotation. This function takes a renaming level m, a success continuation ξ, and a failure continuation ϕ_1, and invokes ξ with the failure continuation ϕ that was current at the time the current relation was entered. ϕ thus replaces ϕ_1 as the current failure continuation when cut is encountered. This approach is possible only when a dynamic binding mechanism exists, since when exiting a relation the cut continuation of its caller must become the current cut continuation.

The auxiliary function *mgu* returns the most general unifier of two value sequences, or *fail* if no unifier exists. It is standard except for the added specification that failure continuations unify only with unbound variables. A semantics that allowed failure continuations to unify with themselves would be reasonable. However, since intensional functions cannot be tested for equality, and failure continuations do not have an extensional representation in our semantics, additional machinery (such as a store semantics or pairing continuations with unique tags) would be required for continuation equality to be meaningful.

The equality goal involves unification of value sequences obtained by down converting, renaming and substituting two term sequences. If the unification succeeds

$unparse$: Term $\rightarrow$ Subst $\rightarrow$ N $\rightarrow$ ((Rel $\times$ Den* $\times$ Den) + $\{fail\}_\perp$)

$$unparse\,T\sigma n = \begin{cases} (R, \vec{\delta}, \delta'), & \text{if } T \mathrel{\mathsf{E}} \mathit{Structure} \text{ and } T{\downarrow}^{(n)} \equiv pattern \\ & \text{or } T \mathrel{\mathsf{E}} \text{Var},\ \delta = \sigma\{T{\downarrow}^{(n)}\} \neq unbound, \text{ and } \delta \equiv pattern; \\ fail, & \text{otherwise.} \end{cases}$$

where $pattern = (\,.\,, \langle \leftarrow, (\,.\,, \langle (\,.\,, \langle R, \vec{\delta} \rangle), (\,.\,, \langle \delta', [\,] \rangle) \rangle) \rangle)$

ans : Subst $\rightarrow$ Asubst

$$ans\,\sigma[\![V]\!] = \begin{cases} unbound, & \text{if } \sigma\{V^{(0)}\} = unbound; \\ \sigma\{V^{(0)}\}{\uparrow}, & \text{otherwise.} \end{cases}$$

alt : Rel $\rightarrow$ Den* $\rightarrow$ Gden

$alt\,\langle\rangle\vec{\delta}m\xi\phi\sigma\beta = \phi$

$$alt\,\{(\vec{\delta}_1, \delta_2, \pi, n') : \rho\}\vec{\delta}m\xi\phi\sigma\beta\gamma n = \begin{cases} \pi n \xi' \phi' \{\sigma\sigma'\}\beta'\gamma'\{n+1\}, & \text{if } \sigma' \neq fail \text{ and } n' = 0; \\ \phi'\gamma n, & \text{otherwise.} \end{cases},$$

where $\xi' = \lambda\phi\sigma\beta_1.\xi\phi\sigma\beta$, $\phi' = alt\,\rho\vec{\delta}m\xi\phi\sigma\beta$, $\sigma' = mgu(\vec{\delta}^{(m)}\sigma, \vec{\delta}_1{}^{(n)})$,
$\beta' = \beta[\,!\ \mapsto n]$, and $\gamma' = \gamma[n \mapsto \langle(\langle\rangle, \mathtt{true}\downarrow, \lambda m\xi\phi_1.\xi\phi, 0)\rangle]$

$delete$: Rel $\rightarrow$ Rel $\rightarrow$ Den* $\rightarrow$ (N $\rightarrow$ Ans) $\rightarrow$ Dcont $\rightarrow$ N $\rightarrow$ Ans

$delete\,\langle\rangle\rho\vec{\delta}\psi\kappa = \psi$

$delete\,\{(\vec{\delta}_1, \delta_2, \pi, n') : \rho_1\}\rho\vec{\delta}\psi\kappa n =$
$$\begin{cases} \kappa\{\rho \mathbin{\S} \langle(\vec{\delta}_1, \delta_2, \pi, n) \mathbin{\S} \rho_1\rangle\}\sigma'\{n+1\}, & \text{if } \sigma' \neq fail \text{ and } n' = 0; \\ delete\,\rho_1\{\rho \mathbin{\S} \langle(\vec{\delta}_1, \delta_2, \pi, n')\rangle\}\vec{\delta}\psi\kappa n, & \text{otherwise.} \end{cases}$$

where $\sigma' = mgu(\vec{\delta}, \{\delta_2 : \vec{\delta}_1\}^{(n)})$

$find$: Rel $\rightarrow$ N $\rightarrow$ Rel

$$find\,\{(\vec{\delta}_1, \delta_2, \pi, n') : \rho\}n = \begin{cases} \rho, & \text{if } n = n'; \\ find\,\rho n, & \text{otherwise.} \end{cases}$$

Figure 4. Auxiliary functions (continued)

with substitution σ', then the success continuation is invoked with, among other arguments, the composition of σ' with the current substitution. Otherwise, the fail continuation is invoked.

Evaluation of conjunctions is accomplished by evaluating the first subgoal with a success continuation that evaluates the second subgoal if the first succeeds. Similarly, disjunctions are handled with a failure continuation that tries the second goal if the first fails. This failure continuation is closed over (contains) the current substitution σ and dynamic database β, while the success continuation created by and does not close over σ or β. It is this that gives substitutions and dynamic databases their dynamic character. In the absence of database assertion, failure continuations may close over the indexing level, allowing indexing levels to be reused upon backtracking. [4] However, since assertions are not normally retracted upon backtracking, indexing levels cannot be reused in the current semantics.

In a `call` goal, the value bound to the given variable is *up converted* to obtain the term to be called. Up conversion translates the value upon which renaming and substitution have been performed back to a syntactic form. It fails if the value being converted contains a failure continuation, since continuations do not have an extensional representation. If up conversion succeeds, the resulting term is coerced to a goal by the auxiliary function *goal*. This coercion may also fail, since not all terms represent goals. If the goal coercion succeeds, the resulting goal is evaluated. The call fails if the variable is unbound or if up conversion or goal conversion fails. Unlike North's treatment of cut [17], a cut in a called goal cuts across the call (installs the caller's cut continuation). To make the cut continuation of the called goal be the fail continuation of the call, all that is required is to update the global database as in γ' of the auxiliary function *alt*.

The argument to `asserta`, `assertz` and `retract` must either be a clause or a logic variable bound to a value of a form that could, with one exception, be up-converted to form a clause. The exception is that values representing "terms" in the clause's head may contain continuations. The auxiliary procedure *unparse* returns the clause's relation identifier R and values representing the "head" and goal of the clause. If the clause is indicated by a variable binding, then these components must be extracted from a value whose structure is that of values constructed by a Prolog term of the form `[:-,[R|T],G]`, where `R` is bound to the relation name, `T` is bound to the head term list, and `G` is bound to the goal. This value is composed of a number of product and sequence constructions, as indicated by *pattern*. The notation $\delta \equiv$ *pattern* indicates that the value δ matches the form of the pattern, and instantiates the pattern variables.

For `asserta` and `assertz`, a new triple representing the asserted clause is then added to the relation by functionally extending the global database. They differ only in that `asserta` inserts the triple at the head of the relation sequence, while `assertz` inserts it at the tail.

For `retract`, the *delete* auxiliary function is called to search the relation sequence indicated by R for a clause that matches the values derived from the head and body terms. *delete* is called with a special success continuation and the failure function $\delta\gamma$. The success continuation κ receives a new relation sequence ρ, a new substitution σ' resulting from the match, and a new indexing number n. κ invokes the success continuation ξ of the retract goal with the new failure continuation ϕ', a new substitution formed by composing the old substitution with σ', the dynamic database, a global database extended to record the deletion, and the new indexing level n. The deletion is recorded by entering the current index n in the deleted 4-tuple. The fail continuation ϕ' uses *find* to locate this tuple and then attempts another deletion on the remaining tuples of the relation.

Because ϕ' retrieves the current database binding, $\gamma\iota$, of the relation, it is possible to delete a tuple added by `assertz` since the last deletion. This conforms makes it straightforward to maintain queues using `assertz` and `retract`. However, the set of rules that may possibly match a goal is determined at the time the goal is called (and is not affected by assertions or retractions that may occur during evaluation of the goal or between success of the goal and subsequent backtracking into the goal). Lindholm and O'Keefe refer to this as the *logical* view of dynamic Prolog code, and argue that it is superior to immediate update and implementational views. [14]

Matching is performed by unifying the head and tail values with the values in the relation triples after the latter have been renamed at a new level n. If unification succeeds, k is invoked with a relation sequence that has had the matching triple removed, the substitution σ' from the unification, and a new indexing level $n+1$. If the end of the relation sequence is reached without finding a match, the failure function ψ is invoked.

The first argument of a `protect` goal is a term that must either be a relation identifier (after the necessary projection to *Const*, *Functor*, and *Relid*) or a variable denoting a relation identifier. The second argument, a subgoal, is evaluated with an extended dynamic database that maps R to a new index n, and an extended global database that maps n to a copy of the current binding of R. The success continuation ξ' of the goal evaluation is closed over the old dynamic database so that the previous binding of R is restored if control returns with success. The previous binding is also restored if control backtracks out of the goal evaluation by invocation of the failure continuation ϕ, since failure continuations are always closed over the dynamic environment.

The `cfc` goal unifies the result of down converting, renaming and substituting the given variable V with the result of injecting the current failure continuation ϕ into the Den domain. If the current substitution does not bind V, or binds it to an unbound variable, this unification will succeed and the resulting substitution σ' will bind V to a value obtained by injecting ϕ into Den. In this event the success continuation ξ is passed the composition of σ' with the current substitution. Otherwise, the goal fails.

The `cut_to` goal fails unless the binding $\sigma V^{(m)}$ of the given variable V in the current substitution is an injected failure continuation. If it succeeds, ξ is invoked with the failure continuation obtained by projecting $\sigma V^{(m)}$ into the Fcont domain.

The `csc` goal is similar to `cfc` except that the given variable is bound to the result of injecting a success continuation into the Den domain. This success continuation, ξ', is the same as the current success continuation, ξ, except that it is closed over the current dynamic database, β. Following successful unification, the given subgoal is invoked with the success continuation ξ and an extended substitution.

The `throw` goal is similar to the `cut_to` goal. However, a success continuation, rather than a failure continuation, is extracted from the given variable. The extracted continuation is then invoked by passing it the current failure continuation, among other arguments.

In this semantics, goals fail in the event of errors. This avoids the need to specify an error handling mechanism. In some cases, such as the failure of up conversion, it would be appropriate for an implementation to raise an error if a suitable mechanism were available.

5. Conclusion

The depth-first control regime of Prolog and most other logic programming languages supports concise and efficient solutions to problems for which depth-first search is appropriate. However, such a control regime often proves to be an obstacle in solving problems requiring more sophisticated control behavior. We have addressed this problem by proposing extra-logical extensions of Prolog for (1) reifying the current success or failure continuation and binding it to a variable, (2) invoking a reified continuation (installing it in place of the current success or failure continuation), and (3) protecting named relations so that their database bindings are automatically restored on backtracking.

Our reified continuations are "first-class" objects that may be stored in the database. These extensions may be used directly, or to implement higher-level operations that abstract recurring patterns of control behavior. They allow variations on breadth-first search and multiprogramming to be implemented efficiently without resort to meta-level programming techniques.

We have presented a denotational semantics of Prolog that includes these extensions. Expressing the reification of success and failure continuations is straightforward. However, for reasons of abstraction and efficiency we do not feel it is appropriate for continuations to have an external representation or for equality of continuations to be defined. Since traditionally Prolog data have external representations and a well defined notion of equality, the addition of reified continuations has ramifications at a number of points in the semantics. It is also notable that a dynamic database allows the cut operation to be expressed without an additional mechanism.

First-class reified continuations allow the control structure to branch, forming a tree, rather than being limited to a stack. This is a consequence of the ability to invoke them after control has left their dynamic context. When first-class continuations are provided in functional and imperative languages it is also necessary, in general, to abandon traditional stack implementation techniques and heap allocate control information. The high cost typically associated with heap storage management has limited acceptance of first-class continuations. However, recent implementation experience with Scheme [5,12] and ML [1] has demonstrated that this cost may often be reduced to a few percent. This is accomplished by optimizations such as the use of a stack at most times, with control information being copied to the heap only when first-class continuations are created, and by particularly efficient garbage collection techniques. Current work is demonstrating that such techniques may be used to provide efficient implementations with logic continuations.

In logic programming languages first-class failure continuations pose an additional implementation concern: when a failure continuation is invoked it is necessary for logic variable bindings to be restored to the values they had when the continuation was created. With a dynamic database mechanism, protected database values must similarly be restored when a continuation is invoked. Though this is most simply accomplished, as in our semantics, by "deep binding" substitution and dynamic database information, "shallow binding" may be required for efficient implementation. In previous work we have developed a state-space mechanism that restores shallow bound logic variable bindings when first-class continuations are invoked [10,19], and similar techniques may be used to restore database bindings.

Acknowledgements

We thank Dan Friedman and Stan Jefferson for their comments on this paper. Kevin Likes, Venkatesh Choppella, and Mike Ashley contributed to the design and implementation of Conlog.

References

[1] Appel, A., and Trevor, J., "Continuation-passing, closure-passing style," *Proc. of the Sixteenth Annual ACM Symp. on Principles of Programming Languages* (1989), 292–302.

[2] Arbib, B., and Berry, D. M., "Operational and denotational semantics of Prolog," *Journal of Logic Programming,* Vol. 4 (1987), 309–329

[3] Boyer, R. S., and Moore, J. S., "The sharing of structure in theorem proving programs," in *Machine Intelligence 7,* B. Meltzer and D. Michie, eds., Edinburgh University Press (1972), 101–116.

[4] de Bruin, A., and de Vink, E.P., "Continuation Semantics for PROLOG with Cut," Technical Report IR-160, Vrije Universiteit Amsterdam, August 1988.

[5] Clinger, W., Hartheimer, A., and Ost, E., "Implementation strategies for continuations," *Proc. of the 1988 ACM Conference on LISP and Functional Programming,* 124–131.

[6] Clocksin, W. F., and Mellish, C. S., *Programming in Prolog,* Springer-Verlag (1981).

[7] Debray, S. K., and Prateek, M., "Denotational and operational semantics for Prolog," *J. Logic Programming,* Vol. 5 (1988), 61–91.

[8] Felleisen, M., "Transliterating Prolog into Scheme," Computer Science Dept. Technical Report No. 182, Indiana University (1985).

[9] Gallaire, H., and Lasserre, C., "Metalevel control for logic programs," in K. L. Clark and S.-A. Tärnlund, eds., *Logic Programming,* Academic Press (1982), 173-185.

[10] Haynes, C., "Logic continuations," *The Journal of Logic Programming,* Vol. 4 (1987), 157–176.

[11] Haynes, C., and Friedman, D. P., "Embedding continuations in procedural objects," *ACM Trans. Programming Languages and Systems,* Vol. 9, No. 4 (1987).

[12] Hieb, R., Dybvig, R. K., and Bruggeman, C., "Representing Control in the presence of first-class continuations," to appear in *Proc. of the SIGPLAN 90 Symposium on Programming Language Design and Implementation.*

[13] Jones, N., and Mycroft, A., "Stepwise development of operational and denotational semantics of PROLOG," *Proc. Symposium on Logic Programming,* Atlantic City (1984).

[14] Lindholm, T., and O'Keefe, R., "Efficient implementation of a defensible semantics for dynamic Prolog code," *Proc. 4th Int'l Conf. on Logic Programming,* J.-L. Lassez, ed., MIT Press, 1987.

[15] Nicholson, T., and Foo, N., "A denotational semantics of Prolog," *ACM Trans. on Programming Languages and Systems,* Vol. 11 (1989), 650–665.

[16] Nilsson, N., *Principles of Artificial Intelligence,* Tioga Publishing Co. (1980).

[17] North, N. D., "A denotational definition of Prolog," National Physical Laboratory Report DITC 106/88, Teddington, Middlesex, United Kingdom, 1988.

[18] Rees, J., and Clinger, W., "The revised^3 report on the algorithmic language Scheme," *Sigplan Notices,* Vol. 21, No. 12 (1986), 37–79.

[19] Salter, R., and Ashley, J., "A revised state space model for a logic programming embedding in Scheme," BIGRE Bulletin, July 1989, 1–8.

[20] Salter, R. M., and Haynes, C. T., "Continuation-based control operators for logic programming," Computer Science Dept. Technical Report No. 293, Indiana University (1989).

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

[22] Wand, M., "A semantic algebra for logic programming," Computer Science Dept. Technical Report No. 148, Indiana University (1983).