A SEMENTIC ALGEBRA FOR LOGIC PROGRAMMING

Mitchell Wand
Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405 USA

TECHNICAL REPORT NO. 148

A SEMANTIC ALGEBRA FOR LOGIC PROGRAMMING

by

Mitchell Wand

August, 1983

Research reported herein was supported in part by the National Science Foundation under grant number MCS 79-04183.

A Semantic Algebra for Logic Programming

Mitchell Wand

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405 USA

August, 1983

Abstract

We use the tools of denotational semantics to analyze and give a semantics for some common operations in logic programming. We then consider the direct execution of the semantics, using the programming language Scheme.

This Material is based on work supported by the National Science Foundation under grant MCS79-04183.

1 Introduction

A denotational semantics provides a syntax-directed transduction from the phrases of a programming language to the elements of some domain of meanings. Because the transduction is syntax-directed, the meaning of any phrase in the language may be calculated as a combination of the meanings of its immediate constituents.

Thus the domain of meanings becomes an algebra, with these combinations as its operations. We refer to this kind of algebra as a *semantic algebra*. Semantic algebras may be used to describe languages [Mosses 82; Clinger, Friedman, and Wand 82], or they can be used for compiler generation [Wand 82a,b].

We create a semantic algebra for a language by analyzing the operations induced by the way in which the language combines phrases to produce new meanings. It may be, for example, that a typical source-language combination is really composed of several smaller combinations, which deserve study in their own right. We may then make the smaller combinations independent operations in the algebra. In this way, we can consider modifications to the original language which are suggested by the semantics.

In this paper, we shall turn this machinery to the study of Logic Programming. We shall create a semantic algebra suitable for logic programming. (Throughout this paper, we shall use the word Prolog to mean a "vanilla" language for logic programming, rather than for any particular implementation).

By combining the atomic operations in new ways, we will find some useful extensions to conventional logic programming. In particular, we introduce a technique for local procedures (predicates) in order to conserve global name space, and we introduce a method for explicitly handling control-flow operations which subsumes and extends the "cut" operation.

The resulting semantics can be executed by embedding it in the programming language Scheme. In this way, Prolog becomes a subset of Scheme. Scheme programs can call Prolog programs, and vice versa, in a fairly convenient way. Our embedding is an alternative to that of LOGLISP [Robinson & Sibert 81]. LOGLISP emphasizes the conception of Prolog as an engine for logical deduction, whereas we choose to emphasize the "computing machine" model of Prolog.

Furthermore, our embedding, unlike many Lisp implementations of Prolog, is not an interpreter. Rather than having an interpreter which parses and inteprets some representation of Prolog code, the embedding produces Scheme functions which directly manipulate the objects of the Prolog semantics. This eliminates the overhead in interpretation due to parsing.

The remainder of the paper is organized as follows: in the next section, we present a short example to give a flavor of the embedding. In Section 3, we define the domains used by the semantics. Section 4 defines the operations of the algebra. Sections 5 and 6 illustrate how the embedding may be used to augment the power of logic programming, and Section 7 gives some conclusions.

2 Example: Append

In Prolog, the predicate for concatenating two lists is written as follows:

```
Append([],Y,Y).
Append([A|D],U,[A|V]) :- Append(D,U,V).
```

We would write this procedure as follows:

```
(define-checked Append
   procedure
   (Alt
       (match '(nil y y) succeed)
       (match '((a . d) u (a . v))
           (call (lambda () Append) '(d u v)))))
```

Here we are defining an identifier **Append** to denote an object of type **procedure**. This type corresponds to Prolog clauses or procedures. This procedure is built by combining two elementary procedures, each built by the operation **match**, using the combinator **Alt**. **match** builds an elementary procedure using a pattern and a **command**, which is executed if the pattern matches the actual parameter to which it is applied. In the first clause, if the pattern matches, the elementary command **succeed** is executed, corresponding to the empty body in the first clause of the Prolog version. In the second clause, a `call` command is to be executed, recursively invoking the procedure **Append**.

Except for the details of concrete syntax, we hope that this code looks sufficiently like Prolog to be comprehensible to a Prolog programmer. (The quote marks and the lambdas will be explained in section 5). One could easily write a macro processor to translate from conventional Prolog syntax to this target.

The next two sections will be devoted to explaining the plan of this embedding. We first define the domains over which the operations of the algebra are defined. Then we define and explain each of these operations in some detail. We then go on to show how this embedding allows for easy extension of the Prolog language.

3 Domains

3.1 Data

The first key domain to be considered is the domain *Subst* of *substitutions*. Substitutions are the real "values" manipulated by Prolog programs.

We regard a substitution as a map from a finite set of variable symbols to finite trees. If t is a tree and σ is a substitution, we write $t\sigma$ for the effect of performing the substitution on the tree t. It is useful to think of the variables which appear in trees as registers, and of substitutions as register files or assignments to be made to such registers. This assignment is performed by the composition operator, which is given by:

$$comp = \lambda\sigma\sigma_1.\lambda v.v \in dom\,\sigma \rightarrow (v\sigma)\sigma_1,\, undefined$$

Note that the domain of the substitution is significant: the substitution $\sigma_1 = \{X \mapsto Z, Y \mapsto Y\}$ is distinct from $\sigma_2 = \{X \mapsto Z\}$, since $comp\,\sigma_1\tau \neq comp\,\sigma_2\tau$, where $\tau = \{Y \mapsto h(U)\}$.

3.2 Control

We next consider control structure. The standard way of describing control structure in denotational semantics is by the use of *continuations*, which are functions which abstract the idea of "the rest of the program." A program phrase which returns a value of type X will have a semantics involving a continuation of type $X \rightarrow Ans$. In this case the continuation is a function which, given the value of the phrase (an element of X), gives the final answer for the entire program (an element of Ans).

In Prolog one has two possible continuations for a phrase, corresponding to success and failure. One might say that nothing is returned on failure. The Prolog store, however, is not undone on failure, so, following the standard technique of denotational semantics, we regard a failure as returning the (possibly modified) store. Hence a failure continuation is modelled as a function from states to answers:

$$K_f = S \rightarrow Ans$$

A succeeding computation may be regarded as passing to its continuation a substitution (its result), a failure continuation (a resumption in case a failure backs into it), and the modified store:

$$K_s = K_f \rightarrow Subst \rightarrow S \rightarrow Ans = K_f \rightarrow Subst \rightarrow K_f$$

(Here we have used the standard trick of "Currying" a function: that is, we replace a 2-argument function $f : A \times B \rightarrow C$ by a function $g : A \rightarrow (B \rightarrow C)$, which, given an element of A, returns a function from B to C. $\rightarrow$ associates to the right, so the parentheses are optional. In making this transformation, we can choose the order in which arguments are supplied. Here we have chosen to make K_f the first argument; this simplifies the definition of the domain of commands below).

3.3 Embedding the metalanguage

In this paper, we shall be discussing two enterprises simultaneously. The first is the development of a denotational, continuation-oriented semantics for Logic Programming. The second is the study of how that semantics can be executed by embedding it in the programming language Scheme [Steele & Sussman 78].

Scheme is a dialect of Lisp with lexical binding, functions as first-class citizens, and applicative-order reduction. In this way our semantics will become executable. One can think of Scheme as a syntactically sugared lambda-calculus. Most of the examples in this paper are actually written using a type-checking preprocessor for Scheme [Wand 83].

Since Scheme functions are first-class citizens, we will model most functions as functions. Thus, for example, a success continuation is modelled as a function which takes a failure continuation and returns a function which takes a substitution and returns an answer. The one exception to this convention is that we shall use Scheme's global store to represent the Prolog store, rather than passing the Prolog store as a parameter. Thus a failure continuation is modelled not as a function of one argument (the store), but rather as a function of no arguments. Changes to the store will be modelled by using the store operations in Scheme.

3.4 Meanings of Program Phrases

We can now consider the meanings of program phrases in Prolog. There are two basic kinds of phrases in Prolog: *commands* and *procedures*.

A command is typified by a procedure call. It takes a success continuation, a failure continuation, a substitution (the current values of the local variables), and the current store. From this it can compute the final answer for the whole program, since the control context is embedded in the continuations. Thus the meaning of a command is a function in the domain

$$\begin{aligned} Cmd &= K_s \rightarrow K_f \rightarrow Subst \rightarrow S \rightarrow Ans \\ &= K_s \rightarrow K_s \end{aligned}$$

Commands merely transform substitutions. To initialize substitutions for use by commands, we need procedure phrases. A procedure is typified by a Prolog clause. It takes a success continuation, a failure continuation, a term (the actual parameter), and the current store. From this information it completes the computation. Thus we introduce the domain

$$Proc = K_s \rightarrow K_f \rightarrow Term \rightarrow S \rightarrow Ans$$

In the next section we will see examples of commands and procedures.

4 Operations

4.1 Simple commands and combinations

We shall begin by describing some simple commands. As suggested earlier, we will write down their semantics in Scheme: in fact, all the code segments are taken from a running implementation.

The simplest command merely succeeds. Given a success continuation, a failure continuation, and a substitution, it passes the failure continuation and the substitution unchanged to its success continuation. In the lambda-calculus, this could be written as:

$$\lambda \kappa_s \kappa_f \sigma . \kappa_s \kappa_f \sigma$$

which is equivalent to

$$\lambda \kappa_s . \kappa_s$$

In the Scheme version, this could be written as follows:

```
(define-checked succeed
        command
        (lambda (ks) ks))
```

Failure is almost as simple. *fail* absorbs its two continuations and its substitutions, and then returns its failure continuation:

$$fail = \lambda \kappa_s \kappa_f \sigma . \kappa_f$$

The Scheme version is similar, except that it then applies the failure continuation to a tuple of no arguments to produce an answer:

```
(define-checked fail
        command
        (lambda (ks)
            (lambda (kf)
                (lambda (s) (kf)))))
```

Local variables may be introduced by the command `(local I)`, which always succeeds, passing on an extended substitution:

```
(define-checked local
        (-> (seq literal) command)
        (lambda (var)
            (lambda (ks)
                (lambda (kf)
                    (lambda (s)
                        ((ks kf) (ext s var)))))))
```

We can combine commands with the function *Seq*, which takes two commands and produces their sequential combination. $Seq\, c_1 c_2$ executes c_1 in a success continuation which then executes c_2:

$$Seq\, c_1 c_2 = \lambda \kappa_s . c_1 (c_2 \kappa_s)$$

In the Scheme version, `Seq` may be defined as follows[1].

---

[1] In the second line of the definition, `seq` is used to declare the type of argument lists in Scheme. Thus this definition declares `Seq` to be a function of two arguments, both commands, producing a command.

```
(define-checked Seq
   (-> (seq command command) command)
   (lambda (cmd1 cmd2)
       (lambda (ks)
           (cmd1 (cmd2 ks)))))
```

Corresponding to the way in which *Seq* builds up success continuations, *Alt* builds up failure continuations: $Alt\, c_1 c_2$ executes c_1; if it or its successors fail, then c_2 is executed.

$$Alt\, c_1 c_2 = \lambda \kappa_s \kappa_f \sigma . c_1 \kappa_s (c_2 \kappa_s \kappa_f \sigma) \sigma$$

One typically applies *Alt* to procedures (clauses) to build up sets of alternative clauses, but it could just as well be applied to commands, in place of the semicolon in Prolog. We shall see later how this facility can be useful.

4.2 Procedures

We have seen simple commands and methods for combining them. We next turn to the commands that actually get some work done: procedure calls. In the conventional picture of Prolog, when one tries to instantiate an atomic formula `p(t1 ... tn)` one attempts to unify it with each of the clauses in the database associated with the predicate symbol `p`. In our picture, the clauses for `p` have been combined by `Alt` into a single procedure that can be called by supplying it with a success continuation, a failure continuation, and the values of the actual parameters.

Thus, associated with a procedure call is an operation `call` in the semantic algebra which takes a procedure and a term and produces a command:

$$call = \lambda pt . \lambda \kappa_s \kappa_f \sigma . p(??) \kappa_f (t \sigma)$$

If the procedure fails, then the call should fail to its failure continuation κ_f. The actual parameter is evaluated by instantiating it in the current substitution σ.

The success continuation ?? for the procedure is more subtle, since it involves the protocol between a procedure and its caller. To see how this should be done, consider the procedure call `p(f(X),Y)` in the substitution $\{X \mapsto g(Z), Y \mapsto Y\}$. The evaluated actual parameters to the procedure are $(f(g(Z)), Y)$. It is the job of the procedure to figure out an instantiation for `Z` and `Y`. We therefore make the procedure return a substitution for `Z` and `Y`; it becomes the caller's job to use this information to update `X` and `Y` appropriately. Luckily, *comp* is just what is needed here. Hence the semantics of *call* can be given by:

$$call = \lambda pt.\lambda \kappa_s \kappa_f \sigma . p(\lambda \kappa'_f \sigma' . \kappa_s \kappa'_f (comp\, \sigma \sigma')) \kappa_f (t\sigma)$$

When the procedure succeeds, the updated substitution $comp\, \sigma\sigma'$ is passed to the success continuation κ_s, along with the failure continuation κ'_f supplied by the procedure body, so that later failure will back into the procedure body.

In order to give the Scheme version of this, we need to make one more small change. It will turn out to be convenient to make the first argument to `call` not a procedure, but a function of no arguments returning a procedure. With this change, we get:

```
(define-checked call
   (-> (seq
           (-> (seq) procedure)         ; a function of no arguments
           term)
        command)
   (lambda (proc term)
       (lambda (ks)
           (lambda (kf)
               (lambda (s)
                   (((proc)
                         (lambda (kf1) (lambda (s1)
                                          ((ks kf1)(comp s s1)))))
                        kf)
```

```
                (subst term s)))))))
```

With `call` done, we can now turn to building primitive procedures. The operation for building primitive procedures is called `match`. Given a term (the pattern) and a command, `match` builds a clause, much as in conventional Prolog. The clause accepts a success continuation, a failure continuation, and an evaluated actual parameter. If the parameter unifies with the pattern, then the command is executed with an appropriate substitution. If not, the clause fails.

Thus, a first version of *match* might be:

$$match = \lambda t_1 \alpha . \lambda \kappa_s \kappa_f t_2 . unifiable\, t_1 t_2 \rightarrow \alpha \kappa_s \kappa_f (mgu\, t_1 t_2), \kappa_f$$

This picture is considerably complicated by the necessity to rename variables. It is necessary not only to rename the variables before the match but also to keep track of that renaming during command execution. The second step is necessary so that the procedure can report a substitution to its caller in terms of the variables which actually appeared in the parameter. The situation is still more complicated, because the returned trees may contain variables which may clash with variables appearing in the caller's substitution.

To take care of standardizing-apart, we introduce a function `rename-vbles` which takes *three* parameters: two trees t_1 and t_2, and a function f of 2 arguments. The variables in t_2 are renamed to be different from those in t_1, resulting in a new term t'_2. The renaming is recorded in a substitution σ_2 such that $t_2 \sigma_2 = t'_2$. The value of *rename-vbles* $t_1 t_2 f$ is then $f t'_2 \sigma_2$. The use of such functions f is a convenient mechanism for procedures which return multiple results.

To avoid variable clashes between procedure and caller, we assume that $mgu\, t_1 t_2$ reports a substitution whose domain is the union of the free variables in t_1 and t_2, and in which *all* variables are bound to new variables which appear nowhere else in the program. (This can easily be done using `gensym`, of course). This corresponds to allocating local registers for each procedure invocation. We also assume that `local`, as defined above, is similarly modified.

We can now write the definition of `match`, which is probably the most complicated definition in this paper:

$$\begin{aligned} \mathit{match} = {} & \lambda t_1 \alpha . \lambda \kappa_s \kappa_f t_2 . \\ & \quad \mathit{rename\text{-}vbles}\, t_1 t_2 \\ & \quad\quad \lambda t'_2 \sigma'_2 . \mathit{unifiable}\, t_1 t_2 \rightarrow \\ & \quad\quad\quad \alpha(\lambda \kappa'_f \sigma' . \kappa_s \kappa'_f (\mathit{comp}\, \sigma'_2 \sigma'))\kappa_f(\mathit{mgu}\, t_1 t_2), \\ & \quad\quad\quad \kappa_f \end{aligned}$$

This definition differs from the previous one in two ways. First, it does the renaming, as discussed above. Secondly, on successful completion of the command α, it uses the result of α to instantiate the renaming σ_2 and hence to report a substitution whose domain is the set of free variables of the actual parameter t_2.

Again, an example will help. Let us consider again the procedure call `p(f(X),Y)` in the substitution $\{X \mapsto g(Z), Y \mapsto Y\}$. The evaluated actual parameters to the procedure are $(f(g(Z)), Y)$. Assume that `p` is bound to the procedure $\mathit{match}\,(f(Z), h(U))\alpha$ for some command α. When this procedure is applied,

- *rename-vbles* renames the variables in $(f(g(Z)), Y)$, passing to its third argument the renamed term $(f(g(Z1)), Y1)$ and the substitution $\{Z \mapsto Z1, Y \mapsto Y1\}$ as t'_2 and σ'_2 respectively.
- The unifier then produces the substitution $\{Z \mapsto Z2, Z1 \mapsto Z2, U \mapsto U2, Y1 \mapsto h(U2)\}$.
- Now, let us imagine that α succeeds, producing the substitution $\{Z \mapsto j(k(V)), Z1 \mapsto$

$j(k(V)), U \mapsto k(V), Y1 \mapsto h(k(V))\}$. (Note that the nature of composition forces the relationships between Z and $Z1$ and between U and $Y1$ to be maintained, thus getting some of the nature of call-by-reference).

- This result σ' will then be composed onto σ'_2, producing the substitution $\{Z \mapsto j(k(V)), Y \mapsto h(k(V))\}$, which is passed to the success continuation κ_s along with the failure continuation κ'_f for backtracking.
- The operation *call* then composes this returned substitution with the substitution in the caller, yielding $\{X \mapsto g(j(k(V))), Y \mapsto h(k(V))\}$ which is passed, along with the failure continuation, to the caller's success continuation.

This completes the procedure call and return. Note that the mechanism is essentially that of call-by-value-result.

5 Examples: Scoping

To illustrate how one might write programs using these operations, we recall the example of append:

```
(define-checked Append
   procedure
   (Alt
       (match '(nil y y) succeed)
       (match '((a . d) u (a . v))
           (call (lambda () Append) '(d u v)))))
```

We now look at this code in more detail. `Append` is a procedure, produced by the alternation of two atomic procedures (clauses). The first is the standard termination clause. The first argument to `match` is quoted, since Scheme evaluates its arguments. The second argument is not quoted: what should be passed to `match` is not the atom `succeed` but its value, the command which we defined earlier. The second alternative is the usual recursion clause, in which we invoke `Append` recursively. Since `call` needs not a procedure, but a function producing a procedure, we make the second argument `(lambda () Append)` instead of just `Append`.

One would execute this code by writing something like:

```
((((call (lambda () Append) '(x y ('a 'b 'c))) init-succeed) init-fail) init-subst)
```

where the last three identifiers denote appropriate initial continuations and an initial substitution. Note that there is no interpreter in the traditional Lisp sense of a piece of code which parses a piece of syntax (or list structure) and takes appropriate actions to some global registers. Instead, each procedure and command is a function which itself performs an action on the semantics objects which are supplied to it as parameters. The combinators, such as `Alt` combine these functional objects into larger functional objects. Thus the parsing overhead of interpreters is eliminated.

Digression: Why didn't we let the first argument to `call` be just a procedure? Then, after all, we could have written, more simply:

```
(define-checked Append
   procedure
   (Alt
       (match '(nil y y) succeed)
       (match '((a . d) u (a . v))
           (call Append '(d u v)))))
```

Unfortunately, the fact the Scheme evaluates its arguments, which serves us well elsewhere, betrays us here: When this form is evaluated, it tries to look up `Append` and fails (essentially because no `lambda` intervenes), rather than getting a recursive definition as desired.

In some Schemes [Fessenden *et. al.* 83], we could have successfully written the recursion successfully as:

```
(define-checked Append
   procedure
(letrec ((Append
          (Alt
              (match '(nil y y) succeed)
              (match '((a . d) u (a . v))
                   (call Append '(d u v))))))
        Append))
```

but this would have defeated our goal of making the Scheme code look as much like Prolog as possible. **End of Digression.**

The fact that procedures are first-class citizens enables us to decouple predicates from the store. This allows the use of local predicates instead of help functions which are globally scoped but only locally useful. As an example, consider the problem of reversing a list. One common way of doing this is as follows:

```
Rev(X,Y) :- Rev1(X,[],Y).
Rev1([],Y,Y).
Rev1([A|D],X,Y) :- Rev1(D, [A|X], Y).
```

This has the unfortunate property of using an extra name, `Rev1`, for a procedure that is useful only inside `Rev`. We can avoid this by binding `Rev1` lexically:

```
(define-checked Rev
   procedure
   (letrec
       ((Rev1
            (Alt
                (match '(nil z z) succeed)
                (match '((a . d) y z)
                    (call (lambda nil Rev1) '(d (a . y) z))))))
       (match '(x y) (call (lambda nil Rev1) '(x nil y)))))
```

Here we have used the local binding facility `letrec` of Scheme to create a local recursive (hence `letrec` rather than `let`) procedure `Rev1`, which is then called by `Rev`.

The convention in Scheme, as in Algol or Pascal, is that identifiers are first searched for in the lexical or static chain; if an identifier is not lexically bound then it is looked up in the global store. This would seem to be a painless extension to Prolog.

One may wonder if we can retain the useful idea of adding clauses to the data base. This can also be done easily. We state it as a Scheme macro:

```
(add-clause procname proc) =>
    (if (globally-defined? procname)
        (change! procname (Alt procname proc))
        (change! procname proc))
```

The process of adding a clause is essentially that of changing the old procedure to the alternation of the old procedure and the new clause.

6 Controlling Control Structure

By embedding Prolog in the executable metalanguage Scheme, we can continue to write in Prolog, but at the same time, we can conveniently extend the language by writing new command- or procedure-valued functions.

As an example of this idea, let us consider the problem of seizing of the control structure of a Prolog program. This is the job of the cut operator; more elaborate mechanisms have been proposed [Clark & Tärnlund 1982]. In the metalanguage, we can do this easily. Let us define a procedure-valued function *bind-failure*:

$$\textit{bind-failure} = \lambda f \kappa_s \kappa_f . f \kappa_f \kappa_s \kappa_f$$

or, in the Scheme embedding:

```
(define-checked bind-failure
   (-> (seq (-> (seq kf) procedure))
       procedure)
   (lambda (f)
       (lambda (ks)
           (lambda (kf)
                (((f kf) ks) kf)))))
```

A typical use of **bind-failure** would be in code of the form `(bind-failure (lambda (failpt)` *proc*`))`. `(lambda (failpt)` *proc*`)` produces a function from failure continuations to procedures, and the entire form produces a procedure. When this procedure is called, the function is applied to the current failure continuation, and hence *proc* will be executed with `failpt` bound to the failure continuation at procedure entry.

How might this value of `failpt` be used? To use it, we introduce the function `fail-to`:

```
(define-checked fail-to
   (-> (seq kf) command)
   (lambda (kf)
       (lambda (ks)
           (lambda (kf1)
               (lambda (s) (kf))))))
```

This function absorbs a failure continuation and produces a command which, when executed, fails not to the current failure continuation but to the failure continuation which was originally supplied to it.

As an example, let us consider a one-shot version of append:

```
(define-checked Append1
   procedure
   (bind-failure
       (lambda (exit)
           (Alt
               (match '(nil y y) (Alt succeed (fail-to exit)))
               (match '((a . d) u (a . v))
                   (call (lambda () Append1) '(d u v)))))))
```

Here `exit` is bound to the failure continuation on entry to the procedure. If backtracking fails back to the point where `Append1` succeeded, the `Alt` (like ";")

calls the `fail-to`, which causes the entire call to `Append1` to fail. Hence, `Append1` will only give one answer; if it is backed into, it will fail.

This is an idiom for cut. In general, one would write

```
(define-checked Foo
   procedure
   (bind-failure
      (lambda (cutpt)
         .....  (Alt cmd (fail-to cutpt)) ....)))
```

Like `Alt`, `bind-failure` can be used for commands as well as procedures. We can use this property to write negation:

```
(define-checked not
    (-> (seq cmd) cmd)
    (bind-failure
        (lambda (exit)
            (Alt
                (Seq cmd (fail-to exit))
                succeed))))
```

This is a function from commands to commands. `(not` *cmd*`)` tries to execute *cmd*. If it succeeds, it fails to the failure continuation `exit`, thus causing `(not` *cmd*`)` to fail. If *cmd* fails, then `Alt` tries the `succeed` causing the entire command to succeed.

7 Conclusions

We have given a semantic analysis of some common features of Logic Programming. By doing so, we have created a framework in which to discuss problematical features of logic programming languages, such as negation, assertion, and the cut operator. The framework also suggests some possible extensions.

The resulting semantics can be embedded into the executable meta-language

Scheme. By doing so, one retains the clarity of traditional logic programming, while gaining the additional power of the metalanguage for better control over binding and over the run-time mechanisms of the language.

This embedding allows one to write Prolog *in* Scheme. Prolog programs become a special case of Scheme programs, and can be called by other Scheme programs. Similarly, Prolog programs can call Scheme programs to do portions of their computation whenever that is convenient. All this is done not by the traditional embedding technique of interpretation, but by building Scheme objects which directly manipulate the underlying semantic objects, thus eliminating interpreter overhead due to parsing.

References

[Clark & Tärnlund 82]

Clark, K.L., and Tärnlund, S.-A. *Logic Programming,* Academic Press, New York, 1982.

[Clinger, Friedman, and Wand 82]

Clinger, W., Friedman, D.P., and Wand, M. "A Scheme for a Higher-Level Semantic Algebra," presentation at US-French Seminar on the Application of Algebra to Language Definition and Compilation, Fontainebleau, France, 1982; proceedings to appear.

[Fessenden *et. al.* 83]

Fessenden, C., Clinger, W., Friedman, D.P., and Haynes, C. "Scheme 311 Version 4 Reference Manual," Indiana University Computer Science Department Technical Report No. 137, February 1983.

[Mosses 82]

Mosses, P. "Abstract Semantic Algebras!" *Proc. TC-2 Working Conference: Formal Description of Programming Concepts II* (D. Bjorner, ed.) (Garmisch-Partenkirchen, 1982), preliminary proceedings, pp. 63–88.

[Robinson & Sibert 81]

Robinson, J.A., and Sibert, E.E. "LOGLISP: an Alternative to PROLOG," in *Machine Intelligence 10* (J.E. Hayes, D. Michie, & Y-H Pao, eds), Ellis Horwood Limited, Chichester, and John Wiley, New York, (1981), 399–419.

[Steele & Sussman 78]

Steele, G.L. and Sussman, G.J. "The Revised Report on SCHEME," Mass. Inst. of Tech. Artif. Intell. Memo No. 452, Cambridge, MA (January, 1978).

[Wand 82a]

Wand, M. "Semantics-Directed Machine Architecture" *Conf. Rec. 9th ACM Symp. on Principles of Prog. Lang.* (1982), 234–241.

[Wand 82b]

Wand, M. "Deriving Target Code as a Representation of Continuation Semantics" *ACM Trans. on Prog. Lang. and Systems 4*, 3 (July, 1982) 496–517.

[Wand 83]

Wand, M. "A Semantic Prototyping System," June, 1983.