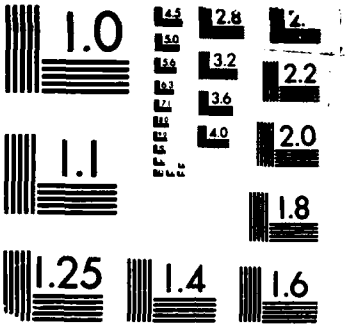
AD-A193 643 UNIFICATION AND SET-VALUED FUNCTIONS FOR FUNCTIONAL AND 1/1
LOGIC PROGRAMMING..(U) NORTH CAROLINA UNIV AT CHAPEL
HILL DEPT OF COMPUTER SCIENCE F S SILBERMANN ET AL.
UNCLASSIFIED SEP 87 TR87-039 N00014-86-K-0680 F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART

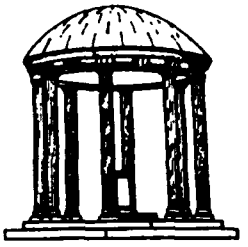
BUREAU OF STANDARDS-1963-A

Unification and Set-Valued Functions for Functional and Logic Programming

TR87-039

September 1987

Frank S.K. Silbermann and Bharat Jayaraman



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Unification and Set-valued Functions for Functional and Logic Programming†

Frank S.K. Silbermann
Bharat Jayaraman

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27514

Abstract

We propose a new approach to the integration of functional and logic languages, based on a theory of unification and set-valued functions. A set-valued function maps a tuple of input sets into an output set. We describe a language called **Setlog** which illustrates this approach, and gives its model-theoretic, fixed-point, and operational semantics. The model-theoretic semantics and fixed-point semantics resemble that of Horn logic. The operational semantics uses outermost reduction (for set-valued functions) and unification (for terms). We establish the correctness of the operational semantics through soundness and completeness proofs.

† This research is supported by a grant DCR-8603609 from the National Science Foundation and contract N 00014-86-K-0680 from the Office of Naval Research.

1. Introduction

The integration of functional and logic programming remains a challenging problem, despite the many proposals that have appeared since Robinson's LOGLISP in the late seventies [BBLM84, DP85, DFP86, F84, GM84, HHT82, JL87, JS86, JSG86, L85, MMW84, R84, R85, RS82, SP85, T84, YS86]. The challenge lies in designing a language that offers semantic simplicity, computational efficiency, programming power and convenience. By semantic simplicity, we mean that the language is amenable to a logical treatment, i.e., model theory and proof theory; by computational efficiency, we mean the language is amenable to a straightforward procedural interpretation; by programming power, we mean the language provides at least the capabilities of first-order functional and Horn-logic languages; and, by programming convenience, we mean that programs model the problem domain as closely as possible.

We offer an approach based on a theory closely related to predicate logic—set-theory. We use definite clauses to define set-valued functions. Functional expresssions denote (possibly infinite) sets of terms. Computation involves proving set-membership assertions, i.e. solving for variables which make the assertions true. The operational procedure uses concepts central to functional and logic programming: outermost reduction and unification of terms.

The language within which we develop these concepts is called **Setlog**. We provide both a model-theoretic semantics and an equivalent fixed-point semantics for Setlog programs, along the lines of van Emden and Kowalski [VK76]. We also provide an operational semantics, based on outermost reduction of functions and unification of terms, and prove its correctness with soundness and completeness proofs. Our approach clearly separates function symbols from constructors. Unification is restricted to first-order terms; function application uses one-way substitution. Setlog can consequently be extended to include higher-order functions without requiring higher-order unification, which is known to be undecidable [H71].

A Setlog program is basically a collection of definite clauses involving set-membership assertions, and may take one of two forms:

1. $term \in f(p_1, \ldots, p_n)$
2. $term \in f(p_1, \ldots, p_n) \leftarrow term \in set\text{-}expression, \ \ldots, \ term \in set\text{-}expression$

Setlog terms, as in logic, are made up of constructors, atoms, and logical variables. The

arguments $p_1, \ldots, p_n$ in the above rules are formal parameter variables, and they must be mutually distinct. Because Setlog distinguishes function symbols from constructors, the formal parameters of a function are distinguished from the logical variables that appear in terms; formal parameters denote sets of terms, whereas logical variables denote single terms. Set expressions are built up of composition of set-valued functions, and the primitive set constructor is { }, which must contain a single term. The symbol ← should be read as "if"; that is, the membership asserted on the left of ← is true if all the memberships asserted on the right of ← are true. The top-level goal is of the form

?- *term* ∈ *set-expression.*

The goal may also be a conjunction of such expressions. The overall computation of a Setlog program proceeds similar to Horn-clause resolution, except that unification is restricted to terms, with outermost reduction used to reduce set-functions.

1.1 Related Work

Several declarative semantic bases integrating functional and logic languages have been proposed: many-sorted Horn-clause logic with equality [GM84], equational logic [F84, GM84, DP85, YS86], and others [MMW84, T84]. Various operational strategies have also been proposed: narrowing [GM84, DP85, R85, YS86], clausal superposition [F84], etc. [JS86, MMW84]. Setlog differs from these various approaches in that it is based on sets, rather than equations or predicates. Because Setlog does not rely on equations, it avoids a computationally expensive unification relative to an equational theory; instead it uses a simple, syntactic unification algorithm.

In a previous paper [JS86], we described a language called **EqL**, which is closely related to Setlog. EqL is a non-deterministic equational programming language combining functional and logic programming. A key idea distinguishing EqL from other equational languages is its separation of constructors from function symbols. A program in EqL has rules of the form

$f(p_1, \ldots, p_n) =$ *expression* **where** *subgoal-equations.*

where each p_i is a formal parameter. The top-level goal is a set of equations. An equation is of the form

$expression_1 = expression_2$

where an expression could have both constructors as well as functions. Because of non-

determinism, functional expressions denoted sets of terms, and therefore the above = predicate is satisfied if the sets denoted by $expression_1$ and $expression_2$ have non-empty intersection. This predicate represents true equality only when relating two deterministic expressions (which denote singleton term sets.). As we investigated the semantics of EqL, we realized that the equational-style syntax was misleading, as our equality was really set-intersection. Improvements to the syntax led to better understanding of the semantic theory, resulting in Setlog.

The need to deal with sets in functional and logic programming has been recognized by several researchers [T81, R84, N85, DFP86, JS86]. Darlington proposes an absolute set abstraction construct for collecting together the solutions of a set of equations [DFP86]; this extends the relative set abstraction construct of Turner [T81]. Naish [N85] surveys a variety of all-solutions predicates proposed for Prolog. Many of these approaches lack a simple declarative semantics. Plaisted and Jayaraman show one way to obtain sets for functional programming without sacrificing their declarative semantics [JP87]; their language does not, however, support logic programming.

The rest of this paper is sectioned as follows: section 2 presents the syntax of Setlog programs, accompanied by examples and its relation to functional and logic languages; sections 3 and 4 give the model-theoretic and fixed-point semantics for Setlog; section 5 is devoted to the operational semantics and ideas for interesting extensions; finally, section 6 presents conclusions and related work. Correctness proofs relating the formal and operational semantics may be found in the appendix.

2. Setlog

2.1 Syntax and Informal Semantics

The syntax of Setlog programs is given by the grammar below. Note that the symbols $\longrightarrow$, |, and ϵ are meta-symbols, and do not belong to Setlog. The symbols $\leftarrow$, $\in$, {, }, (,), ,, and ?- are language-defined tokens. The symbols *setfunctor*, *constructor*, *atom*, and *logical-variable* represent user-defined tokens.

program $\longrightarrow$ *clauses goal*

clauses $\longrightarrow \epsilon$ | *clause clauses*

goal $\longrightarrow$?- *body*

clause $\longrightarrow$ *unit* | *conditional*

unit $\longrightarrow$ *term* $\in$ { *term* } | *head*

conditional $\longrightarrow$ *head* $\leftarrow$ *body*

head $\longrightarrow$ *term* $\in$ *setfunctor*() | *term* $\in$ *setfunctor*(p_1, ..., p_n)

body $\longrightarrow$ *term* $\in$ *setexpr* | *term* $\in$ *setexpr*, *body*

setexpr $\longrightarrow$ { *term* } | *setfunctor*() | *setfunctor*(*setexprs*)

setexprs $\longrightarrow$ *setexpr* | *setexpr*, *setexprs*

term $\longrightarrow$ *atom* | *logical-variable* | *constructor*(*terms*)

terms $\longrightarrow$ *term* | *term*, *terms*

Note that we distinguish between a *setfunctor* and a *constructor*. The former is used for the name of a set-valued function, the latter for a data constructor. A *ground term* is a term without any variables; similarly a *ground set expression* is a set expression without any logical variables. In the rule for *unit*, $p_1, \ldots, p_n$ represent *formal parameters*. Our convention is that *logical-variables* must begin with an uppercase letter, *formal parameters* begin with a lowercase letter, and non-numeric *atoms* are quoted (as in LISP).

We use the constructor cons for constructing binary trees, as in LISP. Lists are a special form of binary trees, and we write them using the `[...]` notation, e.g. `[1,2,3]`. `[]` stands for the empty list, and is regarded as an atom. Thus the list `[1, 2, 3]` is represented as `cons(1, cons(2, cons(3, [])))`. We use the notation `[H | T]`, as in Prolog [CM81], to refer to a non-empty list, with head `H` and tail `T`. Thus, `[H | T]` $\equiv$ `cons(H, T)`.

We also permit the notation {...} to define a set literal, e.g. {1,2,3}. This is just syntactic sugar for a set-function, say `s123()`, defined as follows:

```
1 ∈ s123()
2 ∈ s123()
3 ∈ s123()
```

A Setlog program consists of two kinds of definite clauses: unit and conditional. Both unit and conditional clauses assert set membership. Examples of well-formed unit clauses are:

```
1 ∈ s123()

[H|T] ∈ f(set1, set2)
```

Examples of well-formed conditional clauses are:

$$\texttt{X} \in \texttt{f(s1)} \leftarrow \texttt{[X | T]} \in \texttt{g(h(\{25\}, \{X\}))}$$

$$\texttt{[X | T]} \in \texttt{f(s1, s2)} \leftarrow \texttt{X} \in \texttt{g(s1)}, \texttt{T} \in \texttt{h(s2)}$$

A clause may introduce logical variables in any singleton-sets used in its body, e.g. {X} in the first example above. A conditional clause states that, for any substitution of set expressions for its formal parameters and any substitution of terms for its logical variables, if all the conditions in its body are true, then the condition in its head is also true.

The procedural interpretation of Setlog clauses is similar to that for Horn clauses, except that both unification and reduction are used in simplifying a goal. In order to reduce a goal

$$term_1 \in \texttt{f}(exp_1, \ldots, exp_n)$$

using a clause

$$term_2 \in \texttt{f(p1,...,pn)} \leftarrow body$$

we unify $term_1$ and $term_2$, and if the unification is successful, we replace all occurrences of the formal parameters p1,...,pn throughout the body of f by $exp_1, \ldots, exp_n$ respectively, and then reduce the goals in the body similarly. Unification can bind logical variables in both the goal and in the clause. A top-level goal is solved if all intermediate goals are solved in this manner.

2.2 Examples

Set Union

$$\texttt{X} \in \texttt{union(s,t)} \leftarrow \texttt{X} \in \texttt{s}$$

$$\texttt{X} \in \texttt{union(s,t)} \leftarrow \texttt{X} \in \texttt{t}$$

$$\texttt{?- X} \in \texttt{union(\{1,2\}, \{2,3\})}$$

A solution would bind X to one of elements from the set {1, 2, 3}. If the implementation is sequential with backtracking, X would be in turn bound to 1, 2, 2, and 3.

There is no need in Setlog to define special cases for when one of the arguments to *union* is the empty set—there is no explicit representation of the empty set in Setlog. If the program clauses do not imply any terms to be members of a set expression's denoted set, then that set expression denotes the empty set.

Set Intersection

```
X ∈ intersect(s,t) ← X ∈ s, X ∈ t
?- X ∈ intersect(union({1,2}, {3}), {2,3,4})
```

In finding solutions to the goal, X would be bound in turn to the elements 2 and 3.

Cross Product

```
[X | Y] ∈ prod(s, t) ← X ∈ s, Y ∈ t
?- answer ∈ prod({1,2}, {3,4})
```

In computing successive solutions to the goal, answer would in turn be bound to each term from the set {[1|3], [1|4], [2|3], [2|4]} .

2.3 Syntactic Sugars

We introduce two syntactic sugars to make programs more readable: First, we let a function definition,

$$term \in f(..., term_i, ...) \leftarrow condition$$

be used as syntactic sugar for

$$term_1 \in f(..., p_i, ...) \leftarrow term_i \in p_i, condition$$

where p_i is the i^{th} formal parameter. Using this sugar, the cross product example becomes

```
[X|Y] ∈ prod(X, Y)
```

For the other syntactic sugar, whenever a term t is used in the goal clause or in a program clause body where a set expression is expected, it is treated as $\{t\}$, the singleton set containing t. For example,

```
X ∈ f(A, B)
```

is syntactic sugar for

```
X ∈ f({A}, {B})
```

where f is a set-valued function, and A and B are logical variables.

2.4 Expressive Power of Setlog

To show the flexibility of the Setlog paradigm, we translate typical functional and logic programs into Setlog. Below are equational definitions for the familiar LISP functions *append* and *reverse*.

$append([\,], y) = y$

$append([h \mid t], y) = cons(h, append(t, y))$

$reverse([\,]) = [\,]$

$reverse([h \mid t]) = append(reverse(t), [h])$

These functions can be translated mechanically into sugared Setlog functions:

```
Y ∈ append([], Y)
[H|Z] ∈ append([H|T], Y)  ←  Z ∈ append(T, Y)
[] ∈ reverse([])
Z ∈ reverse([H|T])  ←  Z ∈ append(reverse(T), [H])
```

Expanding the syntactic sugars, the Setlog definitions become:

```
Y ∈ append(s1, s2)  ←  [] ∈ s1, Y ∈ s2
[H|Z] ∈ append(s1, s2)  ←  [H|T] ∈ s1, Z ∈ append({T}, s2)
[] ∈ reverse(s)  ←  [] ∈ s
Z ∈ reverse(s)  ←  [H|T] ∈ s, Z ∈ append(reverse( {T} ), {[H]} )
```

To see that the translations are correct, note that when input sets are singleton (and ignoring the difference between a term and the singleton set containing that term), then

$term \in expression$

is equivalent to

$term = expression.$

When append is applied to non-singleton sets **s1** and **s2**, the resulting denoted set contains all terms which can be constructed by appending a term in **s2** at the end of a term from **s1**. Similarly, **reverse** applied to a set of lists denotes the set containing every list which can be constructed by reversing a list in the input set.

By translating to Setlog, we gain the ability to run functions backward, as in Prolog. Not only can we solve goals of the form

```
?- answer ∈ append([1,2], [3,4])
```

corresponding to the functional goal

?- $append([1,2], [3,4])$

but we can also solve goals of the form

```
?- [1,2,3,4] ∈ reverse(X)
```

which corresponds to solving the equation

```
?- reverse(X) = [1,2,3,4]
```

The ability to run functions backward integrates functional and logic programming.

Setlog retains the full power of Horn logic programming. Any Horn logic program can be mechanically translated into a Setlog program. Consider the following program, written in DEC-10 Prolog [CM81]:

```
apd([ ], Y, Y).
apd([H | T], Y,[H | Z]) :- apd(T, Y, Z).
rev([ ],[ ]).
rev([H | T],Y) :- rev(T,Z), apd(Z,[H],Y).
?- rev(X,[1,2,3,4]).
```

The converted sugared Setlog program would be:

```
'true ∈ apd([], Y, Y)
'true ∈ apd([H|T], Y, [H|Z]) ← 'true ∈ apd(T, Y, Z)
'true ∈ rev([], [])
'true ∈ rev([H|T], Y) ← 'true ∈ rev(T, Z), 'true ∈ apd(Z, [H], Y)
?- 'true ∈ rev(X, [1,2,3,4])
```

In the next two sections, we develop the formal semantics of Setlog. Our development closely parallels that of Van Emden and Kowalski [VK76] and Lloyd [L84]. The key difference is that a Horn logic program defines predicates operating on terms, whereas a Setlog program defines functions mapping tuples of term sets into new term sets. Setlog promotes a different point of view, more so than a new mathematical theory.

3. Model-theoretic Semantics

A set function maps a tuple of ground term-sets, denoted by its argument set expressions, to a new ground term-set. An interpretation maps a ground set-expression to a set of ground terms, thereby giving meaning to the set functions. An interpretation can be expressed as a set of statements of the form:

$$\{gt_1 \in gse_1,\ gt_2 \in gse_2, \ldots\}$$

where gt_i is a ground term and gse_i is a ground set-expression. Also included are all statements of the form:

$gt \in \{gt\}$

where gt is a ground term — the tautology that any given ground term is a member of the singleton-set containing that ground term.

A program *model* is an interpretation under which all the program clauses are true. For any program P there must exist at least one model, B_P, the model containing

$gt \in gse$

for every possible combination of ground term gt and ground set expression gse. The intersection of all models of a program P is therefore also a model. We call this the least model, M_P.

Theorem 1: Let P be a Setlog program. Then

$M_P = \{gt \in gse \text{ of } B_P \mid gt \in gse \text{ is a logical consequence of program } P\}$.

Proof:

$gt \in gse$ is a logical consequence of P

iff P and $gt \notin gse$ is logically unsatisfiable

iff P has no model consistent without $gt \in gse$

iff $gt \in gse$ is included in all models of P

iff $gt \in gse$ is in M_P.

End of Proof.

The model-theoretic semantics of a Setlog program P is given by M_P.

4. Fixed Point Semantics

We define 2^{B_P} to be the set of all interpretations of a Setlog program P. These interpretations form a complete lattice under the partial order set inclusion, $\subseteq$. The top element is B_P, and the bottom element, $\perp$, is the set containing all statements of the form

$gt \in \{gt\}$

where gt is a ground term, but containing no other statements involving set-functions. The lub (least upper bound) of any set of interpretations is the union of those interpretations. The glb (greatest lower bound) of any set of interpretations is their intersection.

Let P be a program. The mapping T_P: $2^{B_P} \rightarrow 2^{B_P}$ is defined as follows. Let I be an interpretation of P. Then $T_P(I) = \perp \cup \{A \in B_P \mid A \leftarrow A_1, ..., A_n$ is a ground instance of a clause in P and $\{A_1, ..., A_n\} \subseteq I\}$.

We say that I is a model for P

iff for each ground instance $A \leftarrow A_1, ..., A_n$ of each clause in P we have $\{A_1, ..., A_n\} \subseteq I$ implies $A \in I$

iff $T_P(I) \subseteq I$.

Since 2^{B_P} is a complete lattice, and T_P is a continuous mapping, the fixed-point semantics of P, $\text{lfp}(T_P) = T_P{}^{\infty}(\perp)$.

The following theorem establishes the equivalence of the fixed-point and model-theoretic semantics.

Theorem 2: $M_P = T_P{}^{\infty}(\perp)$.

Proof:

M_P

$= \text{glb}(\{I \mid I \text{ is a model for } P\}$

$= \text{glb}(\{I \mid T_P(I) \subseteq I\})$

$= \text{lfp}(T_P)$

End of Proof.

A program goal G is a set of statements, possibly containing logical variables. An **answer substitution** for $P \cup \{G\}$ replaces the logical variables in the goal with ground terms. Let P be a program, G a goal, and θ an answer substitution. θ is a **correct** answer substitution for goal G under program P iff $G\theta$ is a logical consequence of P. Thus, θ is a correct answer substitution for G under P iff $G\theta$ is true with respect to every model of P iff $G\theta$ is true with respect to the least model of G.

5. Operational Semantics

A goal G_i is a list of statements S_1, ..., S_k, where each S_j is of the form

term $\in$ *set-expr.*

Both the term and the set expression may contain logical variables to represent unspecified ground terms.

Let R be some computation rule that selects a statement S_m from goal G_i. We have two cases.

Case 1: Deletion Upon Unification

S_m is of the form $term \in \{term_0\}$.

Let $\theta_{i+1} = \text{mgu}(term, term_0)$.

Let $\gamma_{i+1} = \phi$ (the null substitution).

Let $G_{i+1} = (S_1, ..., S_{m-1}, S_{m+1}, ..., S_k)\theta_{i+1}$.

Case 2: Outermost Reduction

S_m is of the form $term \in f(set\text{-}expr_1, ..., set\text{-}expr_n)$,

and the program contains a clause of the form

$term_0 \in f(p_1, ..., p_n) \leftarrow condition.$

Let $\theta_{i+1} = \text{mgu}(term, term_0)$.

Let $\gamma_{i+1} = \{p_1 \leftarrow set\text{-}expr_1, ..., p_n \leftarrow set\text{-}expr_n\}$.

Let $G_{i+1} = (S_1, ..., S_{m-1}, condition\ \gamma_{i+1}, S_{m+1}, ..., S_k)\theta_{i+1}$.

In either case, we say that

G_i with $\sigma \rightsquigarrow G_{i+1}$ with $\sigma\theta_{i+1}$,

where σ is the partial answer substitution associated with G_i. Note that computation of γ_{i+1}, which binds the formal parameters in an outermost reduction step, is a simple one-way substitution; it requires no unification.

Suppose there is a derivation

$G = G_0$ with $\phi \rightsquigarrow ... \rightsquigarrow G_n$ with $\theta_1 ... \theta_n$.

Then we say

$G = G_0$ with $\phi \rightsquigarrow^* G_n$ with $\theta_1 ... \theta_n$

where ϕ is the empty answer substitution, and $\theta_1, \theta_2, ..., \theta_n$ are the substitutions associated with steps $G_1, G_2, \ldots, G_n$. Suppose

$G \rightsquigarrow^* \Diamond$ with $\theta_1 ... \theta_n$,

where $\Diamond$ is the empty goal. We say that

$\theta = \theta_1 ... \theta_n \uparrow \mathcal{V}(G)$

is a *most general computed answer substitution*, where $\mathcal{V}(G)$ stands for the variables in the top-level goal G.

Soundness and completeness theorems state the equivalence of the operational and model-theortic semantics. Proofs are given in an appendix.

Soundness Theorem: If θ is a most general computed answer substitution, and η is any extension such that $\theta\eta$ binds ground terms to all the variables of $\mathcal{V}(G)$, then $\theta\eta \uparrow \mathcal{V}(G)$ is a correct answer substituion as defined by the fixed-point and model-theoretic semantics.

Completeness Theorem: For any correct answer substitution σ, there exists a derivation of a most general computed answer substitution θ and extension η such that $\sigma = \theta\eta$.

5.2 Example Derivation

We illustrate the operational semantics with the following program for appending two lists, taken from section 2.2.

```
Y ∈ app(s1, s2)  ←  [] ∈ s1, Y ∈ s2
[H|Z] ∈ app(s1, s2) ← [H|T] ∈ s1, Z ∈ app({T}, s2)
?- Ans ∈ app({[1,2]}, {[3,4]})
```

The top-level goal would be written as:

$$\texttt{Ans} \in \texttt{app}(\{[1,2]\}, \{[3,4]\}) \text{ with } \phi$$

The reduction steps leading to a solution are given below:

$\rightsquigarrow$ $[H_1|T_1] \in \{[1,2]\}, Z_1 \in \texttt{app}(\{T_1\}, \{[3,4]\})$
with $\{\texttt{Ans} \leftarrow [H_1|Z_1]\}$

$\rightsquigarrow$ $Z_1 \in \texttt{app}(\{[2]\}, \{[3,4]\})$
with $\{\texttt{Ans} \leftarrow [1|Z_1], H_1 \leftarrow 1, T_1 \leftarrow [2]\}$

$\rightsquigarrow$ $[H_2|T_2] \in \{[2]\}, Z_2 \in \texttt{app}(\{T_2\}, \{[3,4]\})$
with $\{\texttt{Ans} \leftarrow [1 \mid [H_2|Z_2]], H_1 \leftarrow 1, T_1 \leftarrow [2], Z_1 \leftarrow [H_2|Z_2]\}$

$\rightsquigarrow$ $Z_2 \in \texttt{app}(\{T_2\}, \{[3,4]\})$
with $\{\texttt{Ans} \leftarrow [1 \mid [2|Z_2]], H_1 \leftarrow 1, T_1 \leftarrow [2], Z_1 \leftarrow [2|Z_2],$
$H_2 \leftarrow 2, T_2 \leftarrow []\}$

$\rightsquigarrow$ $[] \in \{[]\}, Z_2 \in \{[3,4]\})$
with $\{\texttt{Ans} \leftarrow [1 \mid [2|Z_2]], H_1 \leftarrow 1, T_1 \leftarrow [2], Z_1 \leftarrow [2|Z_2],$
$H_2 \leftarrow 2, T_2 \leftarrow [], Y_3 \leftarrow Z_2\}$

$\rightsquigarrow$ $Z_2 \in \{[3,4]\})$

$$\text{with } \{\text{Ans} \leftarrow [1 \mid [2|Z_2]],\ H_1 \leftarrow 1,\ T_1 \leftarrow [2],\ Z_1 \leftarrow [2|Z_2],$$
$$H_2 \leftarrow 2,\ T_2 \leftarrow [],\ Y_3 \leftarrow Z_2\}$$

$$\rightsquigarrow \quad \phi$$

$$\text{with } \{\text{Ans} \leftarrow [1,2,3,4],\ H_1 \leftarrow 1,\ T_1 \leftarrow [2],\ Z_1 \leftarrow [2,3,4],$$
$$H_2 \leftarrow 2,\ T_2 \leftarrow [],\ Y_3 \leftarrow [3,4],\ Z_2 \leftarrow [3,4]\}$$

The restriction of the final substitution to the variables in the top-level goal yields {Ans ← [1,2,3,4]} as the computed answer.

5.3 Possible Extensions to Higher Order Functions

In Prolog and other Horn logic languages, all variables are bound by unification. Extending these languages to handle higher-order predicates requires higher-order unification, a procedure known to be undecidable [H71]. In contrast, Setlog distinguishes between logical variables and formal parameters, and binds the latter by one-way substitutions. For this reason, we believe Setlog is more amenable to higher-order extension. For instance, it is possible to generalize LISP's `mapcar` function to operate upon set-valued functions and sets of lists, written as follows:

```
[] ∈ mapcar(f, [])
[A|B] ∈ mapcar(f, [H|T])  ←  A ∈ f(H), B ∈ mapcar(f, T)
```

In this example, we have let the formal parameter `f` stand for a function, instead of a set expression.

We do not envision the need to solve for logical variables denoting functions or predicates, since neither first-order logic languages nor higher-order functional languages offer this capability. We feel that one of the chief advantages of the Setlog approach is its potential to deal with higher-order functions without getting into the problem of higher-order unification.

7. Conclusions

We have shown that Setlog has semantic and operational simplicity comparable to pure Prolog (Horn logic), and has equal power. Setlog is similar in that a program consists of a set of definite clauses, and execution searches for substitutions binding terms to logical variables in a goal clause, making the goal a logical consequence of the program.

Whereas Prolog clauses describe predicates defined on terms, Setlog clauses describe functions defined on sets of terms. In Prolog there is no distinction between logical variables and procedural formal parameters. Incorporating higher-order objects into Prolog would require higher-order unification. In Setlog, the distinction between logical variables and formal parameters is marked. A logical variable stands for a term; a formal parameter stands for a set of terms. Logical variables are bound through unification, as in Prolog. The substitutions binding formal parameters to expressions are created by a one-way matching, as in functional programming. As a result, incorporating higher-order functions into Setlog may be easier. Functions can be passed as arguments without requiring unification of functions. We are at present investigating a higher-order extension of Setlog [S87].

Set theory and predicate logic are closely related mathamatical formalisms. The semantics of predicate logic are defined in terms of set theory (the Herbrand universe is a set, as is a Herbrand interpretation and a Herbrand model). On the other hand, the axioms of set theory can be encoded as a theory within predicate logic. Thus, set theory and the predicate logic are two alternative but equivalent theories [Q82], though some problems are expressed more naturally in one than in the other. Because set theory and predicate logic are so closely related, the semantics of Setlog resemble the semantics of Horn clause logic programming [VK76, L84]. However, our semantics defines a system of set-valued functions, instead of general predicates operating upon terms.

As we wished to concentrate on semantics, we have not addressed implementation issues in this paper. However, the reader may see that techniques for sequential Horn logic implementation [WPP77] are adaptable to the implementation of Setlog. It is also possible to exploit both *and* and *or* parallelism [CK81] in the execution of Setlog programs.

Acknowledgements

We thank David Plaisted and Tim Rentsch for reading early drafts and giving many useful comments.

References

[BBLM84] R. Barbuti, M. Bellia, G. Levi, and M. Martelli, "On the Integration of Logic Programming and Functional Programming," In *Internatl. Symp. Logic Programming, IEEE*, Atlantic City, 1984, pp. 160–166.

[CM81] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.

[CK81] J. S. Conery and D. F. Kibler, "Parallel Interpretation of Logic Programs," In *Conf. Functional Prog. Lang. and Comp. Arch., ACM*, 1981, pp. 163–170.

[DFP86] J. Darlington, A.J. Field, and H. Pull, "Unification of Functional and Logic Languages," In DeGroot and Lindstrom (eds.), *Logic Programming, Relations, Functions and Equations*, pp. 37-70, Prentice-Hall, 1986.

[DP85] N. Dershowitz and D. A. Plaisted, "Applicative Programming *cum* Logic Programming," In *1985 Symp. on Logic Programming*, Boston, pp. 54–66.

[F84] L. Fribourg, "Oriented Equational Clauses as a Programming Language." *J. Logic Prog.* **2** (1984) pp. 165-177.

[GM84] J. A. Goguen and J. Meseguer, "Equality, Types, Modules, and (Why Not?) Generics for Logic Programming," *J. Logic Prog.* **2** (1984) pp. 179–210.

[H71] G. Hunter, *Metalogic: An Introduction to the Metatheory of Standard First Order Logic*, MIT Press, Cambridge, 1971.

[HHT82] A. Hansson, S. Haridi, and S.-A. Tärnlund, "Properties of a Logic Programming Language," In *Logic Programming*, Ed. K. L. Clark and S.-A. Tärnlund, Academic Press, 1982, pp. 267–280.

[JL87] J. Jaffar, J.-L. Lassez, "Constraint Logic Programming," In *14th ACM POPL*, pp. 111-119, Munich, West Germany, 1987.

[JP87] B. Jayaraman and D.A. Plaisted, "Functional Programming with Sets," to appear in *Third International Conference on Functional Programming Languages and Computer Architecture*, Oregon, 1987.

[JS86] B. Jayaraman and F.S.K. Silbermann, "Equations, Sets, and Reduction Semantics for Functional and Logic Programming," *In 1986 ACM Symposium on LISP and Functional Programming*, pp. 320-331, Boston, 1986.

[JSG86] B. Jayaraman, F.S.K. Silbermann, and G. Gupta, "Equational Programming: A Unifying Approach to Functional and Logic Programming," In *1986 International Conference on Computer Languages*, pp. 47-57, Miami Beach, October 1986.

[L84] J. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, New York,

1984.

[L85] G. Lindstrom, "Functional Programming and the Logical Variable," In *12th ACM POPL*, New Orleans, Jan 1985, pp. 266–280.

[MMW84] Y. Malachi, Z. Manna, and R. Waldinger, "TABLOG: The Deductive-Tableau Programming Language," In *ACM Symp. on LISP and Functional Programming*, Austin, 1984, pp. 323–330.

[N85] L. Naish, "All Solutions Predicates in Prolog," In *Symp. on Logic Programming*, Boston, 1985, pp. 73-77.

[Q82] W. Quine, *Methods of Logic*, Harvard University Press, Cambridge, 1982.

[R84] J. A. Robinson, "New Generation Knowledge Processing: Syracuse University Parallel Expression Reduction," First Annual Progress Report, December 1984.

[R85] U. S. Reddy, "Narrowing as the Operational Semantics of Functional Languages," In *1985 Symp. on Logic Programming*, Boston, 1985, pp. 138–151.

[RS82] J. A. Robinson and E. E. Sibert, "LOGLISP: Motivation, Design, and Implementation," In *Logic Programming*, Ed. K. L. Clark and S.-A. Tärnlund, Academic Press, 1982, pp. 299–313.

[SP85] G. Smolka, P. Panangaden, "Fresh: A Higher-Order Language with Unification and Multiple Results," Technical Report TR 85-685, Computer Science Department, Cornell University, May 1985.

[S87] F.S.K. Silbermann, "Semantics of Set-Functional Programming," Dissertation Proposal, Department of Computer Science, University of North Carolina at Chapel Hill, March 1987.

[T81] D. A. Turner, "The semantic elegance of applicative languages," In *ACM Symp. on Func. Prog. and Comp. Arch.*, New Hampshire, October, 1981, pp. 85-92.

[T84] H. Tamaki, "Semantics of a Logic Programming Language with a Reducibility Predicate," In *Internatl. Symp. Logic Prog., IEEE*, Atlantic City, 1984, pp. 259–264.

[VK76] M. H. van Emden and R. A. Kowalski, "The Semantics of Predicate Logic as a Programming Language," *J. ACM* **23**, No. 4 (1976) pp. 733–743.

[WPP77] D. H. D. Warren, F. Pereira, and L. M. Pereira, "Prolog: the Language and Its Implementation Compared with LISP," *SIGPLAN Notices* **12**, No. 8 (1977) pp. 109–115.

[YS86] P. A. Subrahmanyam and J-H. You, "Equational Logic Programming: an Extension to Equational Programming," In *19th ACM POPL*, St. Petersburg, 1986, pp. 209-218.

Appendix – Correctness Proofs

We establish here the correctness of the operational semantics. We build up to the soundness and completeness theorems by first proving a series of lemmas. These proofs resemble the proofs for soundness and completeness of Horn clause resolution. The similarity is to be expected, since both Setlog and pure Prolog define a program as a collection of definite clauses. In the case of Setlog, the definite clauses determine the terms included in a set-expression's denoted set; in Prolog the clauses establish the truth of ground predicates. We begin with two definitions:

An *unrestricted* derivation is like the $\rightsquigarrow$-derivations described earlier, except that the unifiers need not be most general.

The *success set* of a Setlog program is the set of all ground conditions A in B_P such that $A \rightsquigarrow^* \Diamond$.

The empty goal $\Diamond$, i.e. the goal with no conditions, is trivially satisfied, and is thus a logical consequence of all Setlog programs. Given a goal G (a set of conditions) containing logical variables, and a Setlog program P, we say that G is a logical consequence of P if for all substitutions η which bind the logical variables of G to ground terms, $G\eta$ is a logical consequence of P.

Lemma 1. Suppose we have a derivation

$G_0 \rightsquigarrow^* G_n$ with θ

then for $i = 0, \ldots n$, G_i contains no formal parameters.

Proof: G_0 is a top-level goal clause. By definition, it cannot contain formal parameters. To prove by induction, we need only show that if G_i contains no formal parameters, than neither does G_{i+1}. We must consider two cases, depending on whether this step is a deletion upon unification, or an outermost reduction. In the case of deletion by unification, we unify two terms, *term* and $term_0$ (neither of which may contain formal parameters),

delete the condition, and apply the unifier to the remaining conditions in the goal clause. Since neither the unifier nor the remaining conditions contain formal parameters, than neither does the new goal clause. In the case of an outermost reduction, θ_{i+1} unifies two terms, *term* and $term_0$, both free of formal parameters, and thus introduces no formal parameters into the new goal. The condition of the program clause *does* contain formal parameters, but these are all removed by the application of γ_{i+1}.

End of Proof.

Lemma 2. Given some goal G_0 under some Setlog program, and a derivation

$G_0 \rightsquigarrow^* G_n$ with θ

If G_n is a logical consequence of the program, then so is $G_0\theta$.

Proof: The proof is by induction on the number of steps in the derivation. It is obviously true with a derivation of zero steps, for which $G_0 = G_n$, and θ is the empty substitution ϕ. Suppose it is true for a derivation of $n-1$ steps. We need show that it is then true for a derivation of n steps. Suppose G_0 consists of the conditions $A_1, ..., A_k$, and that the unifiers of the n-step derivation are $\theta_1, ..., \theta_n$. Since we have

$G_1 \rightsquigarrow^* G_n$ with $\theta_2 ... \theta_n$,

$G_1\theta_2...\theta_n$ is a logical consequence of the program, which is the same as $G_1\theta_1...\theta_n$ (since the last step in forming G_1 from G_0 is application of θ_1. To examine the truth of $G_0\theta_1...\theta_n$, we must consider two cases. If the first derivation step was a deletion upon unification, then $G_0\theta_1...\theta_n$ contains exactly the same conditions as $G_1\theta_1...\theta_n$, but with an additional condition of the form

$term \in \{term\}$,

which is a logical consequence of all Setlog programs. Since $G_1\theta_1...\theta_n$ is a logical consequence of P, then so is $G_0\theta_0...\theta_n$.

Otherwise, the first derivation step was an outermost reduction, applied to a condition in G_0

$term \in f(set\text{-}expr_1, ..., set\text{-}expr_n)$,

using a program clause of the form

$term_0 \in f(p_1, ..., p_n) \leftarrow condition$,

with

$\theta_1 = \text{mgu}(term,\ term_0)$,

$\gamma_1 = \{p_1 \leftarrow set\text{-}expr_1,\ ...,\ p_n \leftarrow set\text{-}expr_n\}$.

$G_0\theta_1$ and G_1 are identical, except that $G_0\theta_1$ has a condition which can be written as

$(term \in f(p_1,\ ...,\ p_n))\gamma_1\theta_1$

where G_1 has

$condition\gamma_1\theta_1$.

The program clause states that any substitution making the latter true also makes the former true. Since $G_1\theta_2...\theta_n$ is a logical consequence of the program, so is $G_0\theta_1\theta_2...\theta_n$.

End of Proof.

Theorem 3: Soundness Theorem. If θ is a computed answer substitution of goal G (under Setlog program P), then $G\theta$ is a logical consequence of the program.

Proof: There is a derivation $G \rightsquigarrow^* \Diamond$ with θ', where $\theta = \theta' \uparrow \mathcal{V}(G)$. Since the empty goal is a logical consequence of all programs, and by Lemma 2, $G\theta' = G\theta$ is a logical consequence of the program, θ is a correct answer substitution.

End of Proof.

Now we develop the completeness proof.

Lemma 4: MGU Lemma. Suppose there is a goal G and a Setlog program P with an unrestricted derivation:

$G \rightsquigarrow^* \Diamond$ with $\theta_1...\theta_n$,

Then there exists a restricted derivation

$G \rightsquigarrow^* \Diamond$ with $\theta'_1...\theta'_n$,

(θ_1, ..., θ_n are most general unifiers) and a substitutions σ such that

$\theta_1...\theta_n\sigma = \theta'_1...\theta'_n\sigma$.

Proof: The proof is by induction on the length of the derivation. Suppose $n = 1$, and

$G \rightsquigarrow \Diamond$ with θ_1,

where θ_1 is not necessarily a most general unifier. Then there exists a most general unifier, θ'_1, such that

$G \rightsquigarrow \Diamond$ with θ'_1,

$\theta_1 \geq \theta'_1$.

and there exists a substitution sigma such that

$\theta_1 = \theta'_1\sigma$.

Suppose the result holds for $n-1$, and there is an unrestricted derivation

$G = G_0 \rightsquigarrow^* \Diamond$ with $\theta_1 ... \theta_n$.

Let θ'_1 be the most general unifier of the terms unified in the first derivation step. Then there exists a substitution ρ such that

$\theta_1 = \theta'_1\rho$.

Defining $G'_1 = G_1\rho$, we see that

$G_0 \rightsquigarrow G'_1$ with θ'_1,

$G'_1 \rightsquigarrow G_2$ with $\rho\theta_2$.

$G'_1 \rightsquigarrow^* \Diamond$ with $\rho\theta_2\theta_3 ... \theta_n$.

By the induction hypothesis, there exists a substitution σ and restricted derivation

$G'_1 \rightsquigarrow^* \Diamond$ with $\theta'_2 ... \theta'_n$.

$\rho\theta_2 ... \theta_n = \rho\theta'_2 ... \theta'_n$.

Combining this derivation with the first step, we have restricted derivation

$G = G_0 \rightsquigarrow^* \Diamond$ with $\theta'_1 ... \theta'_n$.

$\theta_1 ... \theta_n = \theta'_1\rho\theta_2 ... \theta_n = \theta'_1 ... \theta'_n\sigma$.

End of Proof.

Theorem 4. The success set of a Setlog program is equal to its least model.

Proof: The soundness theorem implies that the success set is contained in the least model. We need only show that the success set contains the least model. Consider any arbitrary ground condition A in B_P (A is of the form $term \in setexpr$). By Theorem 2, there exists an integer n such that

$A \in T_P{}^n(\perp)$

(or $T_P{}^n$ for short). We prove by induction on n.

Suppose $n = 0$. Then $A \in \perp$. A is of the form

$$\textit{ground-term} \in \{\textit{ground-term}\}.$$

We have

$$A \rightsquigarrow \Diamond \text{ with } \phi.$$

by deletion upon unification (ϕ is the empty substitution). A is in the success set.

Suppose the result holds for $n - 1$. By the definition of T_P, there exists a ground instance of a program clause

$$B \leftarrow B_1, \ldots, B_k$$

(by ground instance we mean substitution set expressions for the formal parameters with some substitution γ and ground terms for the logical variables with some substitution θ) such that $A = B\gamma\theta$, and

$$\{B_1\gamma\theta, \ldots, B_k\gamma\theta\} \subseteq T_P \uparrow (n-1),$$

$$A \rightsquigarrow (B_1\gamma\theta, \ldots, B_k)\gamma\theta.$$

with an unrestricted derivation step. By the induction hypothesis, each $B_i\gamma\theta$ is in the success set. Because each $B_i\gamma\theta$ is ground, their derivations can be combined into a single derivation

$$(B_1, \ldots, B_k)\gamma\theta \rightsquigarrow^* \Diamond.$$

Thus $A \rightsquigarrow^* \Diamond$, and by the MGU lemma, there exists a restricted derivation, as well, A is a member of the success set.

End of Proof.

Lemma 5. Let A be a condition of the form *term* $\in$ *setexpr*, and a logical consequence of a Setlog program P. Then there is a derivation

$$A \rightsquigarrow^* \Diamond \text{ with } \sigma,$$

$$\sigma \uparrow \mathcal{V}(A) = \phi.$$

The second formula states that σ does not affect any of the logical variables in A, i.e. $A\sigma = A$.

Proof: Suppose A has variables $X_1, \ldots, X_n$. Let $a_1, \ldots, a_n$ be distinct new constants (appearing neither in the program P nor in A). Define $\theta = \{X_1 \leftarrow a_1, \ldots, X_n \leftarrow a_n\}$. Then $A\theta$ is a logical consequence of P. Since $A\theta$ is ground, Theorem 4 states that

$A\theta \rightsquigarrow^* \Diamond$ with σ',

$\sigma' \uparrow \mathcal{V}(A\theta) = \phi$.

Since the a_i do not appear in P nor A, by textually replacing all a_i by X_i in this derivation we obtain the derivation

$A \rightsquigarrow^* \Diamond$ with σ,

$\sigma \uparrow \mathcal{V}(A) = \phi$.

End of Proof.

Theorem 5: Completeness Theorem. Let P be a Setlog program and G a goal. For every correct answer substitution θ there exists a derivation

$G \rightsquigarrow^* \Diamond$ with σ,

and a substitution η such that

$\theta = (\sigma \uparrow \mathcal{V}(G))\eta$.

Proof: Let $G = A_1, ..., A_n$. Since θ is correct, then $G\theta$ is a logical consequence of P. By Lemma 5,

$A_i \rightsquigarrow^* \Diamond$ with σ_i,

$\sigma_i \uparrow \mathcal{V}(A_i) = \phi$.

We can combine these into a single derivation

$G \rightsquigarrow^* \Diamond$ with σ,

$\sigma \uparrow \mathcal{V}(A_i) = \phi$.

Suppose that the sequence of mgu's in the derivation is $\theta_1, ..., theta_m$. Then

$G\theta = G\theta\theta = G\theta\theta_1...\theta_m$.

By the MGU Lemma,

$G \rightsquigarrow^* \Diamond$ with $\theta'_1...\theta'_m$,

such that for some η', $\theta\theta_1...\theta_m = \theta'_1...\theta'_m\eta'$. So, let $\sigma = \theta'_1...\theta'_m$ and $\eta = \eta' \uparrow \mathcal{V}(G)$.

End of Proof.

END
DATE
FILMED
7-88
DTIC