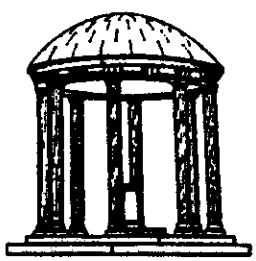
Semantics of Subset-Logic Programming

TR88-053

December 1988

Bharat Jayaraman
David Plaisted



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

Semantics of Subset-logic Programming†

(Summary)

Bharat Jayaraman
David A. Plaisted

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27514
U.S.A.

Telephone: (919) 962-1764
E-Mail: bj@cs.unc.edu

Abstract

Subset-logic programming is a paradigm of programming with subset and equality assertions, and computing with restricted associative-commutative matching and rewriting. The multiple matches arising from a-c matching effectively serve to iterate over the elements of sets, thus permitting many useful set operations to be stated non-recursively. Subset assertions are related to equality assertions by adopting a closed-world assumption (CWA), which is expressed formally by the 'completion' of the program. The principal results are the characterization of the CWA as the least-fixed point of a suitable operator on interpretations, and soundness and completeness of the operational semantics. We also present the formal semantics of the class of stratified subset-logic programs. In contrast to Horn-clause programs with negation, where stratification is used to avoid possible inconsistency arising from completing the program, we introduce stratification here for formalizing the class of closure functions. These functions are useful in defining the smallest set satisfying some property, e.g., various reachability sets. We present declarative and procedural semantics for stratified subset-logic programs. The declarative semantics requires that the auxiliary functions used in defining one closure function in terms of another at the same level be subset-monotonic, but no restriction is placed on the auxiliary functions used from a lower level.

1. Introduction

Subset-logic programming is a paradigm of programming with subset and equality assertions. Our goal in developing this approach [JP87, JN88] has been to provide a more rigorous and efficient way of programming with sets than existing approaches, such as the 'setof' construct of most Prolog systems, which, although very useful in practice, do not provide true sets and are difficult to formalize. A subset-logic program is a collection of two kinds of assertions:

$$f(terms) = expression \qquad \text{and} \qquad f(terms) \supseteq expression.$$

Informally, the declarative meaning of an equality (resp. subset) assertion is that, for all its ground instances, the function f operating on the argument ground terms is equal to (resp. superset of) the ground term denoted by the expression on the right-side. We adopt a closed-world assumption, so that the meaning of a set-valued function f operating on ground terms can be equated to the union of the respective sets defined by the different subset assertions for f. The top-level query is of the form

? *expression*

where *expression* is a ground expression. The meaning of this query is the ground term t such that *expression* $= t$ is a logical consequence of the 'completion' of the program, i.e., augmenting the subset assertions defining some function f with equality assertions that capture the 'collect all' capability of subset assertions.

In earlier papers we described the operational semantics of subset-logic programs in terms of innermost reduction and associative-commutative (a-c) matching [JP87], and also showed how restricted a-c matching can be efficiently compiled into instructions similar to those of the Warren Abstract Machine (WAM) [JN88]. This paper describes the formal semantics of subset-logic programs, and presents new results in two areas.

1. Simple Subset-logic Programs. The principal results here are: (i) a declarative formalization of the closed-world assumption (CWA) in terms of the completion of the program; (ii) a fixed point characterization of the CWA; and (iii) soundness and completeness of the operational semantics. The CWA is computable for subset-logic programs because there is *no* non-membership primitive analogous negation for predicate-logic programs.

2. Stratified Subset-logic Programs. In contrast to Horn-clause programs with negation, where stratification is used to avoid possible inconsistency arising from completing

the program, we introduce stratification here for formalizing the class of *closure functions*. These functions are useful in defining the smallest set satisfying some property, e.g., various reachability sets, dataflow analysis in compilers, etc. In general, stratified subset-logic programs consist of (closure and non-closure) functions that are partitioned into several levels. We provide model-theoretic, fixed-point and operational semantics for stratified subset-logic programs.

To stay within the page limits, we omit proofs of all theorems.

2. Simple Subset-logic Programs: An Informal Introduction

We first specify the syntactic structure of term and expression.

$$term ::= atom \mid variable \mid \phi \mid \{term\} \mid term \cup term \mid constructor(terms)$$
$$terms ::= term \mid term\ ,\ terms$$
$$expr ::= term \mid \{expr\} \mid expr \cup expr \mid constructor(exprs) \mid function(exprs)$$
$$exprs ::= expr \mid expr\ ,\ exprs$$

We will refer to a term as a *set* if it has one of the set constructors, { } or ∪, at its outermost level—these are the only two set constructors—otherwise, we will refer to the term as an *element*. Informally, terms correspond to data objects, and we consider only finite terms. We assume that the constructors ∪ and ϕ obey the following equality theory: $x \cup \phi = \phi \cup x = x$, where x is any set.

Before we present the formal semantics, we first informally describe the two key notions in subset-logic programming: the closed-world assumption, and restricted a-c matching.

2.1 Closed-World Assumption

We incorporate a closed-world assumption in order to relate subset and equality assertions. There are two aspects to our closed-world assumption:

(i) The *collect-all* assumption. If a set-valued expression $\mathbf{s}$ is such that $\mathbf{s} \supseteq \mathbf{s}_1, \ldots, \mathbf{s} \supseteq \mathbf{s}_n$, and it is determined that there are no other known subsets for $\mathbf{s}$ according to the given program, then the collect-all assumption allows us to infer $\mathbf{s} = \cup\, \mathbf{s}_i$.

(ii) The *emptiness-as-failure* assumption. This assumption effectively allows us to discard all failing reductions when collecting the different subsets of a set. In order to deal with failure, we use the device of an undefined element ?, which has the following equality

theory associated with it: $c(\ldots,?,\ldots) = ?$, where c is any element-valued constructor, and $\{?\} = \phi$.

We express both aspects of the closed-world assumption by 'completing' all program assertions by introducing equality assertions. Completion is defined more formally in section 3, but basically there are two cases: In the case of the collect-all assumption, we augment all subset assertions defining a function by an equality assertion that expresses the union of the respective subsets. In the case of the emptiness-as-failure assumption, we add the following equality assertions, so that the value of any expression is either a term or ?, i.e., non-constructors cannot be part of the final answer.

(a) Applying an element-valued (non-constructor) function $\mathtt{f}_e$ to terms that don't match any of the l.h.s. of assertions for $\mathtt{f}_e$ yields ?. Similarly, applying a set-valued function $\mathtt{f}_s$ to terms that don't match any of the l.h.s. of assertions for $\mathtt{f}_s$ yields ϕ.

(b) $\mathtt{f}_e(\ldots, ?, \ldots) = ?$, and $\mathtt{f}_s(\ldots, ?, \ldots) = \phi$.

We illustrate both aspects of the closed-world assumption by the following example. (Note: atoms begin with an uppercase letter, and variables begin with a lowercase letter.)

```
f(Bob) = Mark        m(Bob) = Mary
f(Ann) = Mark        m(Ann) = Mary
f(Mark) = Joe        m(Mark) = Jane

p(x) ⊇ {f(x)}
p(x) ⊇ {m(x)}

anc(x) ⊇ p(x)
anc(x) ⊇ anc(f(x))
anc(x) ⊇ anc(m(x))
```

The collect-all assumption is expressed by augmenting p and anc by the assertions $\mathtt{p(x)} = \{\mathtt{f(x)}\} \cup \{\mathtt{m(x)}\}$ and $\mathtt{anc(x)} = \mathtt{p(x)} \cup \mathtt{anc(f(x))} \cup \mathtt{anc(m(x))}$ respectively.

For example, $\mathtt{p(Bob)} = \{\mathtt{f(Bob)}\} \cup \{\mathtt{m(Bob)}\} = \{\mathtt{Mark}\} \cup \{\mathtt{Mary}\} = \{\mathtt{Mark, Mary}\}$. Similarly, $\mathtt{anc(Mark)} = \{\mathtt{Joe, Jane}\}$ because $\mathtt{p(Mark)} = \{\mathtt{Joe, Jane}\}$, and $\mathtt{anc(f(Mark))} = \mathtt{anc(m(Mark))} = \phi$. The latter holds because $\mathtt{anc(Joe)} = \mathtt{anc(Jane)} = \phi$ by the emptiness-as-failure assumption. Note that $\mathtt{f(Joe)} = \mathtt{m(Joe)} = \mathtt{f(Jane)} = \mathtt{m(Jane)} = ?$ because f and m are element-valued functions, and $\mathtt{anc}(?) = \phi$ because anc is set-valued.

2.2. Restricted A-C Matching

The associative-commutative matching problem may be stated as follows: Given two terms t_1 (possibly non-ground) and t_2 (ground), some constructors of which may be associative-commutative, is there a substitution θ such that $t_1\theta =_{ac} t_2$? ($=_{ac}$ means 'equality modulo the associative and commutative equations'.) This problem was first posed by Plotkin [P72] and has since been studied quite extensively in the literature. We use the notation $\{\mathtt{x} \mid \mathtt{t}\}$ to refer to a non-empty set, one of whose elements is x and the remainder of the set is t. Thus, $\{\mathtt{x} \mid \mathtt{t}\} \equiv \{\mathtt{x}\} \cup \mathtt{t}$. When all set patterns are restricted to the form $\{\mathtt{t1} \mid \mathtt{t2}\}$, where t2 and t2 do not use $\cup$ explicitly, we obtain a more efficient (compilable) matching algorithm, called restricted a-c matching, which we discussed in [JN88]. As shown in [JP87, JN88], the multiple a-c matches of $\{\mathtt{x} \mid \mathtt{t}\}$ provide a convenient and efficient way of iterating over the elements of a set. Here we use a very simple example to convey the basic idea:

$$\mathtt{distr}(\mathtt{x}, \{\mathtt{y} \mid \mathtt{t}\}) \supseteq \{[\mathtt{x} \mid \mathtt{y}]\}$$

When matching an expression such as $\mathtt{distr}(10, \{1,2,3\})$ with the left-side of the assertion defining distr, all three a-c matches are considered, namely, $\{\mathtt{x} \leftarrow 10, \mathtt{y} \leftarrow 1, \mathtt{t} \leftarrow \{2,3\}\}$, $\{\mathtt{x} \leftarrow 10, \mathtt{y} \leftarrow 2, \mathtt{t} \leftarrow \{1,3\}\}$, and $\{\mathtt{x} \leftarrow 10, \mathtt{y} \leftarrow 3, \mathtt{t} \leftarrow \{1,2\}\}$. The right-side of the assertion for distr, namely $\{[\mathtt{x} \mid \mathtt{h}]\}$, is then fully reduced for each of these matches, and the union of the fully reduced results is defined as the value for $\mathtt{distr}(10, \{1,2,3\})$, which in this case = {[10|1], [10|2], [10|3]}. Note: (i) Duplicates must be eliminated when taking this union, but we described in [JP87, JN88] how this check can be avoided for a particular argument when the function 'distributes over union' in this argument. (ii) No assertion is needed for the case when the argument set is empty; $\mathtt{distr}(\mathtt{x}, \phi) = \phi$, by the completion of the program. (iii) The idempotence of $\cup$ is not used in a-c matching, to avoid the possibility of an infinite recursion (see **powerset** example in section 2.3).

Before turning to a more formal account of these ideas, we briefly discuss our confluence assumption.

2.3. Confluence

In order that the non-constructor symbols f introduced by program assertions define functions, we require that the system of equations and subset assertions be confluent. Stated as a syntactic condition, we require that:

(i) the left-hand side of each equality assertion not overlap with any other assertion,

(ii) when set constructors occur in equality assertions, the result should be independent of which one of the potentially many a-c matches is selected.

Note that a subset assertion may overlap with other subset assertions. Other less restrictive conditions are possible, but we shall assume the above conditions, for the sake of specificity. Thus, for example, the following definition of the powerset of a set is legal:

$\texttt{powerset}(\phi) = \{\phi\}$
$\texttt{powerset}(\{\texttt{x} \mid \texttt{t}\}) = \texttt{distr2}(\texttt{x}, \texttt{powerset}(\texttt{t}))$
$\texttt{distr2}(\texttt{x}, \texttt{s}) \supseteq \texttt{s}$
$\texttt{distr2}(\texttt{x}, \{\texttt{y} \mid \_\ \}) \supseteq \{\{\texttt{x} \mid \texttt{y}\}\}$

However, the definition below for set-to-list conversion is not legal.

$\texttt{set2list}(\phi) = [\ ]$
$\texttt{set2list}(\{\texttt{x} \mid \texttt{t}\}) = [\texttt{x} \mid \texttt{set2list}(\texttt{t})]$

In defining the semantics of subset-logic programs in the next section, we assume that program assertions are confluent. In an earlier paper [JP87], we mentioned methods of proving confluence of equational programs with a-c constructors.

3. Semantics of Simple Subset-logic Programs

3.1. Completion

We begin by flattening all expressions so that the arguments of all function calls are terms. Temporary variables are introduced as necessary. For the parent and ancestor assertions mentioned in section 2.1, the flattened form would be

$\texttt{p(x)} \supseteq \{\texttt{e}\}$:- $\texttt{f(x)} = \texttt{e}$
$\texttt{p(x)} \supseteq \{\texttt{e}\}$:- $\texttt{m(x)} = \texttt{e}$
$\texttt{anc(x)} \supseteq \texttt{s}$:- $\texttt{p(x)} = \texttt{s}$
$\texttt{anc(x)} \supseteq \texttt{s}$:- $\texttt{f(x)} = \texttt{e}$, $\texttt{anc(e)} = \texttt{s}$.
$\texttt{anc(x)} \supseteq \texttt{s}$:- $\texttt{m(x)} = \texttt{e}$, $\texttt{anc(e)} = \texttt{s}$.

The general flattened form of an assertion is

H :- B,

where H may be either $\texttt{f}(t) = u$ or $\texttt{f}(t) \supseteq u$, and B is of the form $E_1, \ldots, E_n$, where each E_i is $\texttt{f}_i(t_i) = u_i$. Note: (i) B may be empty, in which case we have unconditional assertion; (ii) f's argument has a single term t rather a sequence of terms, because the latter is

subsumed by the former given a sequence constructor; and (iii) the order of equalities on the r.h.s. reflects the innermost reduction order for expressions. The flattened form of a (ground) query expression will be a sequence of equalities of form $f(t) = x$, where t is a *term* and x is a variable.

1. **Collect-All Assumption.** For each set-valued function $\mathtt{f}_s$ defined by a collection of n subset assertions, $\mathtt{f}_s(t_1) \supseteq s_1$:- B_1, ..., $\mathtt{f}_s(t_n) \supseteq s_n$:- B_n, where B_i stands for the body of each assertion, we add the single equality assertion

$$\mathtt{f}_s(v) = \cup_{i=1,n} \cup \{s_i : (\exists \bar{y}_i)\ v =_{ac} t_i \wedge B_i\}$$

where v is a new variable not appearing in any of the subset assertions defining $\mathtt{f}_s$ and $\bar{y}_i$ are all the variables of the i-th assertion but excluding s_i.

2. **Emptiness-as-Failure Assumption.** For each element-valued function $\mathtt{f}_e$ defined by a collection of n equality assertions, $\mathtt{f}_e(t_1) = u_1$:- B_1, ..., $\mathtt{f}_e(t_n) = u_n$:- B_n, we add

$$\mathtt{f}_e(\mathtt{x}) = ?, \text{ for each } \mathtt{x} \in G - T,$$

where G is the universe of ground terms and T is the set of ground terms obtained by instantiating each of the terms t_1, ..., t_n on the l.h.s of all assertions for $\mathtt{f}_e$. In the case of a set-valued function $\mathtt{f}_s$, defined by equality and subset assertions, we add

$$\mathtt{f}_s(\mathtt{x}) = \phi, \text{ for each } \mathtt{x} \in G - T.$$

Finally, we include $\mathtt{f}_e(\ldots,?,\ldots) = ?$, and $\mathtt{f}_s(\ldots,?,\ldots) = \phi$.

We shall refer to the completion of P as *comp*(P). For the parent-and-ancestor example, the collect-all assumption adds the following equality assertions.

$$\begin{aligned}
\mathtt{p}(\mathtt{v}) = {} & \cup\{\{\mathtt{e}\} : (\exists \mathtt{x})\mathtt{x} =_{ac} \mathtt{v} \wedge \mathtt{f}(\mathtt{x}) = \mathtt{e}\} \ \cup \\
& \cup\{\{\mathtt{e}\} : (\exists \mathtt{x})\mathtt{x} =_{ac} \mathtt{v} \wedge \mathtt{m}(\mathtt{x}) = \mathtt{e}\} \\
\mathtt{anc}(\mathtt{v}) = {} & \cup\{\mathtt{s} : (\exists \mathtt{x})\mathtt{x} =_{ac} \mathtt{v} \wedge \mathtt{p}(\mathtt{x}) = \mathtt{s}\} \ \cup \\
& \cup\{\mathtt{s} : (\exists \mathtt{x}, \mathtt{e})\mathtt{x} =_{ac} \mathtt{v} \wedge \mathtt{f}(\mathtt{x}) = \mathtt{e} \wedge \mathtt{anc}(\mathtt{e}) = \mathtt{s}\} \ \cup \\
& \cup\{\mathtt{s} : (\exists \mathtt{x}, \mathtt{e})\mathtt{x} =_{ac} \mathtt{v} \wedge \mathtt{m}(\mathtt{x}) = \mathtt{e} \wedge \mathtt{anc}(\mathtt{e}) = \mathtt{s}\}
\end{aligned}$$

For the `distr` function shown earlier, the collect-all assumption adds:

$$\mathtt{distr}(\mathtt{v}_1, \mathtt{v}_2) = \cup\{\mathtt{s} : (\exists \mathtt{x}, \mathtt{y})\ (\mathtt{v}_1, \mathtt{v}_2) =_{ac} (\mathtt{x}, \{\mathtt{y} \mid \_\}) \wedge \mathtt{s} = \{[\mathtt{x} \mid \mathtt{y}]\}\}$$

3.2. Model-theoretic Semantics

We define the model-semantics starting from the universe of terms U_P (similar to Herbrand universe) which is the set of *ground terms* of the program P augmented with the undefined

element ?. By P, we refer to the flattened form of the source program. The *base* of interpretations B_P for a program P (similar to Herbrand base) is the set of *unconditional ground* equality and subset assertions derived from the program assertions. A *model* of a program P is an interpretation $I \subseteq B_P$ such that I satisfies every assertion in P, i.e., for all ground instances H^g :- E_1^g, ..., E_n^g, whenever $\{E_1^g, \ldots, E_n^g\} \subseteq I$, we also have $H^g \in I$.

In addition we assume that a model satisfies the following *equality theory for constructors*: (i) associative, commutative, and idempotent properties of $\cup$, (ii) $c(\ldots,?,\ldots) = ?$ for any element-valued constructor c, and (iii) $\{?\} = \phi$. Terms that are not equal by the above equality theory are assumed to be equal only if they are identical.

Definition 1: An assertion A is said to be a *logical consequence* of a program P, denoted $P \models A$, if every model of P is also a model of A.

Definition 2: The *model-theoretic* semantics of P, $M_P = \{ A : comp(P) \models A \}$, where A is an unconditional ground assertion.

Definition 3: Given a program P and a query expression G, we say θ is a *correct answer* substitution, where θ binds all variables in G to ground terms, iff $comp(P) \models G\theta$.

Proposition 1: $comp(P) \models P$.

Proposition 2: If P is terminating, there is a unique model for $comp(P)$.

However, non-terminating subset-logic programs do not have unique models, as illustrated by the following program defining two set-valued functions f and g:

$f(x) \supseteq \{1\}$

$f(x) \supseteq g(x)$

$g(x) = g([x])$

Note that the collect-all assumption adds the following assertion:

$f(x) = \{1\} \cup \cup\{s : g(x) = s\}$.

The above program has an infinite number of models of the form $(\forall x)$ [$f(x) \supseteq \{1\}$ and $f(x) = \{1\} \cup s$, and $g(x) = s$], for any constant set s. However, because of the following proposition, there is a *least model*, which is $(\forall x)$ [$f(x) \supseteq \{1\}$] in this example.

Proposition 3: The intersection of all models of $comp(P)$ is a model.

Theorem 1: $\mathcal{M}_P = \cap\{M : \models_M comp(P)\}$.

The notation $\models_M P$ means M is a model for P.

3.3. Fixed-point Semantics

We characterize the least model as the least fixed-point of an immediate-consequence operator on interpretations, $\mathcal{T}_P$, as follows.

$$\mathcal{T}_P(\mathrm{I}) = \mathcal{R}_P(\mathrm{I}) \cup \mathcal{E}_P(\mathrm{I}),$$

where

$$\mathcal{R}_P(\mathrm{I}) = \{\ \mathrm{H}^g : \mathrm{H}^g \text{ :- } \mathrm{E}_1^g, \ldots, \mathrm{E}_n^g \text{ is a ground instance of a program assertion and } \{\mathrm{E}_1^g \ldots \mathrm{E}_k^g\} \subseteq \mathrm{I}\}$$

$$\mathcal{E}_P(\mathrm{I}) = \{\ \mathtt{f}(t) = \cup\{s : (\exists u)\ \mathtt{f}(u) \supseteq s \in \mathcal{R}_P(\mathrm{I}) \wedge u =_{ac} t\} : \mathtt{f}(u) \supseteq s \in \mathcal{R}_P(\mathrm{I}) \text{ for } \textit{all } u =_{ac} t \text{ and } \textit{all} \text{ subset assertions for } \mathtt{f} \text{ in } P\}$$

We should note that $\mathcal{T}_P$ is defined not on $comp(P)$, but on P supplemented with the equality assertions for the emptiness-as-failure assumption. $\mathcal{R}_P(\mathrm{I})$ derives new unconditional ground assertions from program assertions, and is identical to the transformation rule for predicate-logic programs. $\mathcal{E}_P(\mathrm{I})$ expresses the collect-all assumption by deriving an equality assertion involving $\mathtt{f}(t)$ only when all subsets of $\mathtt{f}(u)$ are known, considering each subset assertion for $\mathtt{f}$ and each term $u =_{ac} t$. Note that are only finitely many $u =_{ac} t$ and also a finite number of subset assertions for $\mathtt{f}$ in P.

Proposition 4: $\mathcal{T}_P$ is continuous.

Theorem 2: $\mathcal{M}_P = \mathcal{T}_P^{\omega}(\perp_{\mathrm{B}})$, where $\perp_{\mathrm{B}}$ is the empty interpretation.

For the example in section 3.2, we have $\mathcal{T}_P \uparrow 0 = \phi$, $\mathcal{T}_P \uparrow 1 = \mathcal{T}_P \uparrow \omega = (\forall \mathtt{x})\mathtt{f}(\mathtt{x}) \supseteq \{1\}$.

3.4. Operational Semantics

As in the fixed-point semantics, we assume that the flattened program P has been supplemented with equality assertions for the emptiness-as-failure assumption. We define a rewriting relation $\rightarrow$ by considering the two mutually exclusive cases in rewriting an innermost expression.

Case 1: Given variants of subset assertions, $\mathtt{f}(t_1) \supseteq s_1$:- B_1, ... $\mathtt{f}(t_n) \supseteq s_n$:- B_n, and a flattened query expression

$$\mathrm{G} \equiv \mathrm{g}_1, \ldots, \mathrm{g}_m,$$

where g_1 is $\mathtt{f}(t) = s$, we define the rewriting relation $\rightarrow$ such that, if matching t with

$$G \rightarrow (B_1\ \theta_{11}), \ldots, (B_1\ \theta_{1k_1}), \ldots, (B_n\ \theta_{n1}), \ldots, (B_n\ \theta_{nk_n}), (g_2, \ldots, g_n)\ \sigma,$$

where $\sigma \equiv \{s \leftarrow \cup_{ij}\ (s_i\theta_{ij})\}$. (Note that s will be a variable.)

Case 2: Given variants of equality assertions, $f(t_1) = u_1$:- B_1, ... $f(t_n) = u_n$:- B_n, and a flattened query expression

$$G \equiv g_1, \ldots, g_m,$$

where g_1 is $f(t) = u$, we define the rewriting relation $\rightarrow$ such that, if matching t with $t_1 \ldots t_n$ yields respectively the substitutions $\theta_{11}, \ldots, \theta_{1k_1}, \ldots, \theta_{n1}, \ldots, \theta_{nk_n}$, then

$G \rightarrow (B_i\ \theta_{ij}), (g_2, \ldots, g_n)\ \sigma$ for some i and j, where $\sigma \equiv \{u \leftarrow (u_i\theta_{ij})\}$.

Definition 4: Given a program P and query G, we say that θ restricted to variables in G is the *computed answer* of a derivation

$$G \equiv G_1 \rightarrow G_2 \rightarrow \ldots \rightarrow G_k \equiv []$$

if $\theta \equiv \sigma_1 \ldots \sigma_k$, i.e., the composition of the substitutions $\sigma_1, \ldots, \sigma_k$ at each step.

The following theorems express the correctness of the operational semantics.

Theorem 3 (soundness): Given a program P and top-level goal sequence G, the computed answer θ is a correct answer.

Theorem 4 (completeness): Given a program P and top-level goal sequence G, if there exists a correct answer θ, then there is a $\rightarrow$-derivation such that θ is the computed answer.

4. Stratified Subset-logic Programs

4.1. Closure Functions

It turns out that, with a suitably enhanced operational procedure, one can compute more information than is defined by the least fixed-point of $\mathcal{T}_P$ (equivalently, the least model, M_P). A trivial example is shown below.

$\mathtt{f(x)} \supseteq \{\mathtt{x}\}$	$\mathtt{g(x)} \supseteq \{\mathtt{x}\}$
$\mathtt{f(x)} \supseteq \mathtt{g([x])}$	$\mathtt{g([x])} \supseteq \mathtt{f(x)}$

The above program is nonterminating (according to the operational semantics of section 3.4), and its least model is $(\forall\, \mathtt{x})\ [\mathtt{f(x)} \supseteq \{\mathtt{x}\}$ and $\mathtt{g(x)} \supseteq \{\mathtt{x}\}]$. That is, in the least model, we know that $\mathtt{f(x)}$ and $\mathtt{g(x)}$ *contain* $\{\mathtt{x}\}$, but we do not know which set $\mathtt{f(x)}$ and $\mathtt{g(x)}$ are *equal* to. In this example, although there are an infinite number of models which contain $\mathtt{f(x)} = \mathtt{g(x)} = \{\mathtt{x},\ [\mathtt{x}]\} \cup s$ for some set constant s, the least model has none of these

assertions. Note also that the nontermination here arises because of a recursive call with an identical argument as an outer call, e.g., `f(x)` = {x} ∪ {[x]} ∪ `f(x)`. Such calls can be detected with the use of a memo-table (or extension-table).

We refer to sets defined cyclically in this manner as *set closures*, and functions defined using such sets as *closure functions*. All closure functions in this language are set-valued functions. A more useful example is the function **reach** below for finding the set of reachable nodes of a graph g, represented as a set of ordered pairs, starting from some given node v.

```
reach(v, g) ⊇ {v}
reach(v, g) ⊇ allreach(adjacent(v, g), g)
allreach({x | _},g) ⊇ reach(x,g)
adjacent(v, {[v, w] | _ }) ⊇ {w}
```

In general, closure functions are useful whenever one is in interested in defining the smallest set satisfying some property, e.g., dataflow analysis in compilers. We require the programmer to identify all closure functions, say through annotations—otherwise the overhead of memo-izing function calls can become excessive. In order to permit the use of closure functions as arguments to non-closure functions and still maintain our ability to define a suitable and fixed-point semantics, we impose three conditions:

(i) We *stratify* all assertions into n levels, where level 1 assertions define only non-closure functions, and each of the remaining levels is divided into two sets of assertions: assertions defining closure functions and assertions defining non-closure functions. In the above example, `adjacent` would be at level one, and the closure functions `reach` and `allreach` would be at level two.

(ii) We require that all closure functions be terminating (after memo-ization). We can relax this requirement, but we will assume it in this paper, for the sake of simplicity in presentation.

(iii) We require that all closure functions at a given level are defined in terms of one another using *subset-monotonic* functions, but may be defined in terms of any (closure or non-closure) function from a lower level. A set-valued function g is said to be *subset-monotonic* in a particular argument iff $s_1 \subseteq s_2 \supset g(\ldots, s_1, \ldots) \subseteq g(\ldots, s_2, \ldots)$. For example, set-difference $x - y$, is *not* subset-monotonic in its second argument, but is subset-monotonic in its first argument.

4.2. Semantics of Stratified Subset-logic Programs

Definition 5: Assuming P_j are all the assertions at level j and P_j^c are just the assertions defining closure functions at level j, M_n defines the *model-theoretic semantics* of a stratified program with n levels, where

for $j > 1$, $\mathrm{M}_j = \mathrm{A}_j \cup \mathrm{C}_j$, where

$$\mathrm{A}_j = \{\ \mathrm{A} : comp(P_j) \cup \mathrm{M}_{j-1} \models \mathrm{A}\},$$

$$\mathrm{C}_j = \{\ \mathbf{f}_{ji}(t) = \cap\{s : (\exists M, u) \models_M comp(P_j^c) \wedge \mathbf{f}_{ji}(u) = s \in M \wedge u =_{ac} t\}$$

$$: \mathbf{f}_{ji} \text{ is a closure function at level } j\},$$

and $\mathrm{M}_1 = \{\ \mathrm{A} : comp(P_1) \models \mathrm{A}\}$,

At level j, A_j defines the semantics of non-closure functions as well as the *subset* assertions for the closure functions that are logical consequences of the program. The *equality* assertion for a closure function $\mathbf{f}_{ji}(t)$ at level j is defined by C_j, and is the intersection of all sets defined for $\mathbf{f}_{ji}(u)$ where $u =_{ac} t$ in the different models M for P_j^c.

To define the fixed-point semantics, let $\mathcal{C}_j$ and $\mathcal{N}_j$ be the transformation operators used to define the semantics of closure and non-closure functions at level j respectively. Their definitions are identical to $\mathcal{T}_P$ of section 3, except that the ordering of intepretations for $\mathcal{C}_j$ is as follows, where $\mathbf{f}$ used below is a closure function.

$\mathrm{I}_1 \sqsubseteq \mathrm{I}_2$ iff $\mathrm{I}_1 \subseteq \mathrm{I}_2 \vee ((\mathbf{f}(\mathrm{t}) = \mathrm{s}_1 \in \mathrm{I}_1) \supset (\mathbf{f}(\mathrm{t}) = \mathrm{s}_2 \in \mathrm{I}_2 \wedge \mathrm{s}_1 \subseteq \mathrm{s}_2))$

The above ordering expresses the requirement that a smaller model should have not only fewer assertions, but also smaller sets.

Proposition 5: $\mathcal{C}_j$ is continuous w.r.t. $\sqsubseteq$.

Definition 6: Assuming P has n levels, the fixed-point semantics is F_n, where

for $j > 1$, $\mathrm{F}_j = \mathrm{F}_{j-1} \cup \mathcal{N}_j^{\omega}(\mathrm{F}_{j-1}) \cup \mathcal{C}_j^{\omega}(\perp_j)$, and

for $j > 1$, $\perp_j = \mathrm{F}_{j-1} \cup \{\mathbf{f}_{ji}(t) = \phi : t \in \mathrm{G},\ i \in 1 \ldots k_j\}$,

and $\mathrm{F}_1 = \mathcal{T}_{P_1}^{\omega}(\perp_B)$

Note that we have effectively assumed that all closure functions are terminating, because of the initialization $\{\mathbf{f}_{ji}(t) = \phi : t \in \mathrm{G},\ i \in 1 \ldots k_j\}$ for each closure function $\mathbf{f}_{ji}$.

Theorem 5: For a stratified program with n levels, $\mathrm{M}_n = \mathrm{F}_n$.

Finally, we sketch the *operational semantics*. We define the $\rightarrow$ relation between pairs of the form <G, T>, where G is a goal-sequence as before, and T is a memo-table, i.e., a

set of assertions of the form $\mathbf{f}(t) = u$, where $\mathbf{f}$ is a closure function, t is a ground term, but u may be non-ground. Initially T = ϕ. Given a pair <G, T>, let the first goal in G, $g_1 \equiv \mathbf{f}(t) = v$. We define <G, T> $\rightarrow$ <G', T'>, as follows.

(i) If $\mathbf{f}$ is a non-closure function, we define G' as in section 3.4. If σ is the computed substitution for v in deriving G', we define T' $\equiv$ T σ.

(ii) If $\mathbf{f}$ is a closure function and there is no assertion of the form $\mathbf{f}(u) = w$ in T for any $u =_{ac} t$, we define G' as before, and T' $\equiv$ (T $\cup$ $\{\mathbf{f}(t) = v\}$) σ.

(iii) If $\mathbf{f}$ is a closure function and $\mathbf{f}(u) = w$ is in T for some $u =_{ac} t$, we define σ $\equiv$ $\{v \leftarrow norm(w, v)\}$, G' $\equiv$ (G - $[g_1]$) σ, and T' $\equiv$ T σ. The term $norm(w, v)$ is the *normalized* form of w with respect to v, i.e., replacing any occurrences v in w by ϕ.

Assuming that the *correct answer* and *computed answer* are re-stated relative to the new semantic definitions, we have the following correctness results.

Theorem 6 (soundness): For a stratified program P and goal G, the computed answer θ is a correct answer.

Theorem 7 (completeness): Given a stratified program P and goal G, if there exists a correct answer θ, there is a $\rightarrow$-derivation such that θ is the computed answer.

For the example defining $\mathbf{f}$ and $\mathbf{g}$ in section 4.1, the top-level query $<\mathbf{f}(1) = \mathbf{v}, \phi>$ has a successful $\rightarrow$-derivation with the computed answer $\{\mathbf{v} \leftarrow \{1\} \cup \{[1]\} \cup \phi\}$. The memo-table at the end of the derivation would be $\{\mathbf{f}(1) = \{1\} \cup \{[1]\} \cup \phi, \quad \mathbf{g}([1]) = \{1\} \cup \{[1]\} \cup \phi\}$.

5. Conclusions

We mention the salient points about the semantics we have presented in comparison with that for Horn-clauses with negation [L87]:

1. Unlike Horn-clauses, the CWA for subset-logic programs is computable, and completion does *not* lead to inconsistency—basically, we do not provide a $\not\supseteq$ or non-membership primitive in the language. The declarative (or model-theoretic) semantics of a subset-logic program P can expressed in terms of the logical consequences of $comp(P)$. Note that models for $comp(P)$ have equality assertions and subset assertions, although our interest at the top-level is in the term that is *equal* to a given query expression.

2. For simple subset-logic programs, the least fixed-point of our $\mathcal{T}_P$ operator characterizes the CWA. The definition of this operator is different from that for Horn clauses in

that it incorporates the closed-world assumption. The operational semantics, based on a-c matching and innermost-first reduction, is correct (sound and complete) with respect to the declarative or fixed-point semantics.

3. Unlike Horn-clauses with negation, stratification in subset-logic programs is needed not to avoid inconsistency but to make possible a fixed-point semantics for *closure functions*. Our semantics require that the auxiliary functions used in defining one closure function in terms of another (at the same level) to be subset-monotonic, and also all closure functions to be terminating—the latter assumption can be relaxed, but we haven't explored this issue in this paper. The operational semantics corresponding to the above definition essentially uses a memo-table in order avoid nontermination from identical nested calls.

References

[JP87] B. Jayaraman and D.A. Plaisted, "Functional Programming with Sets," In *3rd Int'l Conf. on Func. Prog. Lang. and Comp. Arch.*, pp. 194-210, 1987.

[JN88] B. Jayaraman and A. Nair, "Subset-logic Programming: Application and Implementation," In *5th Int'l Logic Prog. Conf.*, pp. 843-858, 1988.

[L87] J.W. Lloyd, "Foundations of Logic Programming," Springer-Verlag, 1987.

[P72] G. Plotkin "Building-in equational theories," In *Machine Intell.*, **7**, pp. 73-90, 1972.

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