Subset Assertions and Negation-As-Failure

Mauricio Osorio
Bharat Jayaraman

Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
Telephone: (716) 645-3194

E-Mail: {`osorio,bharat`}`@cs.buffalo.edu`

Abstract

Subset assertions provide a declarative and natural means for expressing solutions to many problems involving sets. This paper is motivated by the use of subset assertions for formulating transitive closures and solving containment constraints in applications of graph traversal and program analysis. In these applications, circular containment constraints may arise, for which we propose an operational strategy based upon memoization and reexecution of function calls. We provide formal declarative and operational semantics for this class of subset assertions. One of the main technical results of this paper is a succinct translation of subset assertions into normal program clauses [L87] such that the stratified semantics of the resulting normal programs coincides with the declarative semantics of subset assertions. This translation is interesting because the operational semantics of subset assertions appears to be very different from that of normal programs—due to the setof-like capability and the need of reexecution for subset assertions, both of which are absent in normal program clauses. (However this translation is not an acceptable implementation of subset assertions due to its inefficiency.) We also discuss the connection between our proposed declarative semantics and recent approaches such as stable and well-founded semantics.

Keywords: Subset Assertions, Transitive Closures, Memoization, Negation-As-Failure, Stratified Semantics, Well-Founded Semantics

1. Introduction

Subset assertions are of the form[1]

$f(terms) \supseteq expr$

where each variable in *expr* also occurs in *terms* (the syntax *terms* and *expr* is given in section 2). Informally, the declarative meaning of a subset assertion is that, for all its ground instantiations (i.e., replacing variables by ground terms), the function *f* applied to argument terms is a superset of the ground set denoted by the expression on the right-hand side. By incorporating a *collect* capability, the meaning of a set-valued function *f* applied to ground terms is equal to the union of the resulting sets defined by the different subset assertions for *f*. This basic paradigm of subset assertions offers a declarative and succinct means of formulating many set-valued functions, and has been explicated in [JP89, J92].

The motivation for this paper stems from certain natural uses of subset assertions in applications such as graph theory, program analysis, etc. In these applications, one is interested in finding the smallest set that satisfies certain containment constraints. Such problems can be modelled quite directly using subset assertions. It turns out that a natural operational semantics for the resulting programs requires *memoization* [M68, W92] in order to detect circular containment constraints. However, depending upon the syntactic form of these subset assertions, a strict memoization regime does not appear to suffice; rather, a *re-do* capability is needed, i.e., a memoized call may have to reexecuted several times in order to progress towards the least fixed-point. In this sense, the operational strategy is a combination of traditional top-down and bottom-up evaluation.

Although they appear to be syntactically and semantically very different from traditional (predicate) logic programs, we show in this paper that the semantics of subset assertions has closely parallels with the semantics of normal programs [L87], i.e., Horn-like programs where the body of a program clause may have both positive and negative literals. We show a strong result (in section 4.2): subset assertions can be translated into normal programs such that their stratified semantics coincides with the direct semantics of subset assertions (given in section 3). The ability to effect such a translation is significant because the resulting normal programs do not make any explicit use of a *setof* capability, yet they are concise and intuitive. However, these translated programs would not be satisfactory implementations of subset assertions because of their inefficiency.

The model-theoretic semantics of subset assertions is interesting in that their least model is given not in terms of classical intersection of sets, but by intersecting the set-valued terms in corresponding ground atoms in the various models. The other major technical result of this paper is a soundness proof for our proposed operational strategy. Finally, we also discuss the connections between the semantics of subset assertions and recent approaches, such as stable models and well-founded models [GL88, VRS91].

The remainder of this paper is organized as follows: section 2 gives the basic syntax of terms and expressions and explains informally, using examples, the operational semantics of subset assertions; section 3 gives the declarative and operational semantics for subset assertions, and introduces the class of simple-stratified and general-stratified programs; section 4 presents a translation of subset assertions to normal programs and shows the equivalence of the stratified semantics of the resulting normal programs and the declarative semantics of section 3; and, finally, section 5 suggests extensions of the paradigm to non-set-valued functions, which effectively serve as a top-down formulation for dynamic programming problems such as the shortest distance between a pair of vertices in a graph.

---

[1] Our current implementation actually uses the keyword `contains` instead of $\supseteq$, and the keyword `phi` instead of ϕ. It also supports equational assertions, but they are not the focus of this paper and hence we omit them here.

2. Subset Assertions: An Informal Introduction

2.1 Syntax

terms ::= *term* | *term* , *terms*
term ::= *variable* | *constant* | *c*(*terms*) | ϕ | $\{term\backslash set\}$

exprs ::= *expr* | *expr* , *exprs*
expr ::= *term* | $\{expr\backslash expr\}$ | *c*(*exprs*) | *f*(*exprs*)

The symbol c stands for a constructor symbol whereas f stands for a non-constructor function symbol. Terms are built up from constructors and stand for data objects of the language. Our lexical convention in this paper is to begin constants with an lowercase letter and variables with uppercase letters. We refer to a *term* as a *set* if it has one of the set constructors ϕ or $\{\_\backslash\_\}$ at its outermost level. The notation $\{x\backslash t\}$ refers to a set in which x is one element and t is the remainder of the set. We permit as syntactic sugar $\{expr\}$ to stand for $\{expr \backslash\phi\}$, and $\{e_1, e_2, \ldots, e_k\}$ to stand for $\{e_1\backslash\{e_2\backslash \ldots \{e_k\backslash\phi\}\}\}$. To illustrate the set constructor, matching $\{\mathtt{X}\backslash\mathtt{T}\}$ against a ground set {a, b, c} yields three different substitutions, $\{\mathtt{X} \leftarrow \mathtt{a}, \mathtt{T} \leftarrow \{\mathtt{b},\mathtt{c}\}\}$, $\{\mathtt{X} \leftarrow \mathtt{b}, \mathtt{T} \leftarrow \{\mathtt{a},\mathtt{c}\}\}$, and $\{\mathtt{X} \leftarrow \mathtt{c}, \mathtt{T} \leftarrow \{\mathtt{a},\mathtt{b}\}\}$. One should contrast $\{\mathtt{X}\backslash\mathtt{T}\}$ from $\{\mathtt{X}\} \cup \mathtt{T}$. The reason for preferring $\{\mathtt{X}\backslash\mathtt{T}\}$ over $\{\mathtt{X}\} \cup \mathtt{T}$ is that the former allows one to recursively decompose a set into strictly smaller sets. There are only finitely many matches of a set-term $\{\mathtt{X}\backslash\mathtt{T}\}$ against any set constructed from ϕ and $\{\_\backslash\_\}$. Sets built up from $\{\_\backslash\_\}$ are normalized so that they do not contain repeated items.

For example, the intersection of two sets can be defined as follows:

```
intersect({X\_},{X\_})  ⊇  { X }
```

When set patterns occur on the left-hand sides of subset assertions, all matches against these patterns are used in instantiating the corresponding right-hand side expression, and the union of all the resulting sets is taken as the result. To see the use of remainder-sets, suppose we represented a collection of propositional clauses by a set of set of literals, then its set of resolvents can be defined succinctly as follows:

```
resolvents({{X\S1}, {not(X)\S2} \_})  ⊇  {S1 ∪ S2}.
```

Finally, recursive subset assertions are also meaningful and natural (we use Prolog notation [...] for lists):

```
perms(φ)  ⊇  { [ ] }
perms({X\T})  ⊇  distr(X,perms(T))
distr(X,{L\_})  ⊇  { [X|L] }
```

The `perms` function takes a set of elements as input and produces as output the set of lists corresponding to the permutations of these elements. The function `distr` expects a set of lists as its second argument; its result is a set whose elements are constructed by "consing" its first argument to each list in the set denoted by the second argument. The reader is referred to [JP89, J92] for more examples.

2.2 Memoization

One of the more interesting uses of subset assertions is in defining transitive closures. This use reveals the need for an operational semantics with *memoization* [M63, W92]. As an illustration, consider 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`.

```
closure reach
reach({X\_})  ⊇  {X}
reach({X\_})  ⊇  reach(edge(X))
edge(1) ⊇  {2}
```

$\texttt{edge(2)} \supseteq \{1\}$

The above program will be nonterminating when run on an input graph that has a cycle. However, by identifying the definition `reach` as `closure` functions by the following annotation[2]

```
closure reach
```

it is possible to obtain the desired answer even for cyclic graphs—essentially, the infinite loop is avoided through memoization. We use the term *non-closure* function to refer to functions such as `intersect`, etc., of section 2.1, for which no `closure` annotation is provided. Reference [JP89] discusses non-closure functions in detail, but provides only a very brief treatment of closure functions, which are the main topic of this paper.

We illustrate memoization for the query

$? \ \texttt{reach}(\{1\}) = \texttt{Ans}.$

In the derivation below, we assume that all expressions are *flattened* out, to reveal the goal-sequence more clearly (see section 3 for a more precise definition of flattening).

Goal Sequence	**Substitution**	**Memo Table**
$\texttt{reach}(\{1\}) = \texttt{Ans}$		ϕ
$\texttt{edge(1)} = \texttt{T1},$ $\texttt{reach(T1)} = \texttt{S1}$	$\texttt{Ans} \leftarrow \{1\} \cup \texttt{S1}$	$\{\texttt{reach}(\{1\}) = \{1\} \cup \texttt{S1}\}$
$\texttt{reach}(\{2\}) = \texttt{S1}$	$\texttt{T1} \leftarrow \{2\}$	$\{\texttt{reach}(\{1\}) = \{1\} \cup \texttt{S1}\}$
$\texttt{edge(2)} = \texttt{T2},$ $\texttt{reach(T2)} = \texttt{S2}$	$\texttt{S1} \leftarrow \{2\} \cup \texttt{S2}$	$\{\texttt{reach}(\{1\}) = \{1\} \cup \{2\} \cup \texttt{S2},$ $\texttt{reach}(\{2\}) = \{2\} \cup \texttt{S2},$
$\texttt{reach}(\{1\}) = \texttt{S2}$	$\texttt{T2} \leftarrow \{1\}$	$\{\texttt{reach}(1) = \{1\} \cup \{2\} \cup \texttt{S2},$ $\texttt{reach}(\{2\}) = \{2\} \cup \texttt{S2},$ $\texttt{reach}(2) = \{2\} \cup \texttt{S2}\}$
[ ]	$\texttt{S2} \leftarrow \{1\} \cup \{2\}$	$\{\texttt{reach}(\{1\}) = \{1\} \cup \{2\},$ $\texttt{reach}(\{2\}) = \{1\} \cup \{2\}\}$

Note that memo-table look-up occurs in the next to last step. The binding to the variable `S4` is the smallest solution to the equation

$\texttt{S2} = \{1\} \cup \{2\} \cup \texttt{S2}.$

The binding to the variable `Ans` in the top-level query is obtained by composing the substitutions at each step, and is easily seen to be the set $\{1, 2\}$.

(As an optimization, in this example, one need not maintain the partial bindings for the various function calls; it suffices to record just the function calls, in order to detect the loop. When a loop is detected, one simply returns the empty set, ϕ. Such a strategy can be proven to be sound. However, this simple strategy does not suffice for examples to be discussed subsequently.)

---

[2] One might consider all functions to be closure functions and employ static analysis to detect when memoization can be avoided. In this paper we explicitly annotate which functions are to memoized, to remain faithful to our current implementation which does require such annotations.

2.3. Re-Do

For the `reach` example considered above, we need to solve, during a look-up step, an equation of the form $v = s \cup v$, where v is a variable and s is some set. This equation is easily solved with the substitution $v \leftarrow s$. However, for more general programs involving subset assertions, it becomes necessary to solve an equation of the form $p(\ldots, v \cup s, \ldots) = v$, where v is a variable, s is a possibly nonground set, and p is a *subset-monotonic* function, i.e., $s_1 \subseteq s_2 \Rightarrow p(s_1) \subseteq p(s_2)$. This is illustrated by the following very simple program:

```
closure g, h
g(X) ⊇ {10}
g(X) ⊇ h(X)
h(X) ⊇ {20}
h(X) ⊇ p(g(X))
p({X\_}) ⊇ {X, 30}
```

The first four steps in the derivation from the toplevel query `g(100)` are as follows.

Goal Sequence	**Substitution**	**Memo Table**
`g(100) = Ans`		ϕ
`h(100) = S1`	$\mathtt{Ans} \leftarrow \{\mathtt{10}\} \cup \mathtt{S1}$	$\{\mathtt{g(100)} = \{\mathtt{10}\} \cup \mathtt{S1}\}$
$\mathtt{g(100)} = \mathtt{T1}, \mathtt{p(T1)} = \mathtt{S2}$	$\mathtt{S1} \leftarrow \{\mathtt{20}\} \cup \mathtt{S2}$	$\{\mathtt{g(100)} = \{\mathtt{10}, \mathtt{20}\} \cup \mathtt{S2},$ $\mathtt{h(100)} = \{\mathtt{20}\} \cup \mathtt{S2}\}$
$\mathtt{p}(\{\mathtt{10}, \mathtt{20}\} \cup \mathtt{S2}) = \mathtt{S2}$	$\mathtt{T1} \leftarrow \{\mathtt{10}, \mathtt{20}\} \cup \mathtt{S2}$	$\{\mathtt{g(100)} = \{\mathtt{10}, \mathtt{20}\} \cup \mathtt{S2},$ $\mathtt{h(100)} = \{\mathtt{20}\} \cup \mathtt{S2}\}$

At the last step above, we find that the argument to the function `p` is nonground, since the variable `S2` is still undetermined. At this stage, we assume provisionally that $\mathtt{S2} = \phi$ and proceed with the computation, but reconsider this assumption later. Now evaluating the goal $\mathtt{p}(\{\mathtt{10}, \mathtt{20}\} \cup \mathtt{S2})$ with $\mathtt{S2} = \phi$ yields $\mathtt{p}(\{\mathtt{10}, \mathtt{20}\})$ $= \{\mathtt{10}, \mathtt{20}, \mathtt{30}\}$. Thus the revised estimate for `S2` is $\{\mathtt{10}, \mathtt{20}, \mathtt{30}\}$. When a variable such as `S2` has its estimate revised, the goal that used a provisional value of this variable is re-evaluated using the new estimate. Re-evaluating $\mathtt{p}(\{\mathtt{10}, \mathtt{20}\} \cup \mathtt{S2})$ with $\mathtt{S2} = \{\mathtt{10}, \mathtt{20}, \mathtt{30}\}$ yields $\mathtt{S2} = \{\mathtt{10}, \mathtt{20}, \mathtt{30}\}$. Since `S2` has not changed, no further re-evaluation is needed, and the toplevel query has successfully terminated with the answer $\{\mathtt{10}, \mathtt{20}, \mathtt{30}\}$.

In general it is possible that several variables might be nonground when attempting to reduce some subset-monotonic function. All such variables are assumed to be ϕ initially, and their estimates are revised progressively until an overall fixed-point is reached. This example also illustrates the need for subset-monotonic functions: if `p` were not subset-monotonic, the progressive iteration is not guaranteed to reach a fixed-point.

For a more realistic example, consider the formulation of the *reaching definitions* in a program flow graph, which is computed by a compiler during its optimization phase [AU77]:

```
closure in, out, allout
out(B) ⊇ in(B) - kill(B)
out(B) ⊇ gen(B)
```

```
in(B) ⊇ allout(pred(B))
allout({P \_}) ⊇ out(P)
```

where `kill(B)`, `gen(B)`, and `pred(B)` are predefined set-valued functions specifying the relevant information for a given program flow graph and basic block `B`. Since the set-difference operator `-` is subset-monotonic in its first argument, its use in defining the closure function `out` is legal. Because `in(B)` and `out(B)` are defined circularly, memoization is needed to avoid the infinite loop that could result from a naive evaluation strategy. Furthermore, since `out(B)` is defined in terms of `in(B)` via function `-`, there is a need to re-execute certain function calls: When the estimated value of a closure function is revised, the goal that used a provisional value needs to be re-executed (or re-done) until no further re-execution is needed.

3. Semantics of Subset Assertions

We discuss in this section the declarative (i.e., model-theoretic) and operational semantics of closure functions.

In preparation for the semantics, we first flatten all expressions so that the arguments of all function calls are terms. Since all variables range over the universe of terms, this flattened form makes more explicit that the result of an expression must be a term. For example, a clause

```
f(X, {Y \ S}) ⊇ g({h(X) \ k(Y, S)})
```

will be flattened as follows, where `g`, `h`, and `k` are assumed to be non-constructor functions:

```
f(X, {Y \ S}) ⊇ S2    :-  h(X) = T1, k(Y, S) = S1, g({T1 \ S1}) = S2
```

The flattened form of a subset program is in general

Head :- *Body*,

where *Head* is $\mathtt{f}(t) \supseteq u$, where t and u are terms, and *Body* is of the form E_1, ..., E_n, where each E_i is $\mathtt{f}_i(t_i) = x_i$, where $\mathtt{f}_i$ is a user-defined function, t_i is a term, and x_i is a new variable not present on the l.h.s. Notes: (i) The order of equalities on the right-hand side of a flattened subset assertion reflects the *innermost reduction order* for expressions. (ii) *Body* may be empty, in which case we have an unconditional assertion. (iii) Without loss of generality, the argument to a function can be assumed to be a single term t rather a sequence of terms.

In the sequel, we will use the term *closure atom* to refer to an atom of the form $f(t) = s$ or $f(t) \supseteq s$, where f is a *closure* function.

3.1. Declarative Semantics

The declarative semantics for a closure function f applied to any argument t will be defined by taking the intersection of all sets defined for $f(t)$ in the different Herbrand models for f. To appreciate the need for taking such intersections, consider the following trivial program:

```
closure f
f(x) ⊇ {1}
f(x) ⊇ f(x)
```

The different models of this program are as follows:

```
f(x) = {1}
f(x) = {1, {1}}
```

`f(x)` $= \{\texttt{1}, \{\texttt{1}\}, \{\texttt{1}, \{\texttt{1}\}\}\}$
etc.

where, in each model, `x` ranges over the universe of terms. The intended model for closure function `f` is obtained by intersecting the respective sets defined for `f(x)` in the different models. In order to obtain a computable semantics, we *stratify* or partition the program into several levels, as follows.

(i) Level 1 assertions are contrained according to the following syntax.

$f(terms) \supseteq g(term)$
$f(terms) \supseteq set$

where f and g are closure functions at level 1.

(ii) The definition of one closure function in terms of other closure functions at the same level is constrained according to the following syntax.

$f(terms) \supseteq g(lexprs)$
$f(terms) \supseteq lexprs$
$f(terms) \supseteq set$

In the above two cases, f is a closure function at some level j, g is a (not necessarily different) closure function at the same level j, *lexpr* is an expression composed of functions from levels $1, \ldots, j-1$, and *lexprs* is a sequence of one or more *lexpr*. That is, a closure function at level j is directly defined in terms of other closure functions at the same level.

The above class of programs is referred to as **simple stratified programs**. (We discuss general stratified programs in section 3.3.) In the `reach` program shown in section 2.2, the function `edge` would be at level one, and the function `reach` would be at level two.

Definition: Let P be a program. We define P_k as the set of program assertions in P such that the function symbol in the head of the left-hand side of an assertion has level $\leq k$. Sometimes, by abuse of language, we will say "let P_k be a program" to mean that "let P be a program such that $P = P_k$."

We will work with Herbrand Interpretations, where the Herbrand Universe consists of ground terms (including sets) and the Herbrand Base consists of ground (closure and non-closure) atoms†. We further impose the following two restrictions:

1. if $f(t) = s_1 \in$ I and $f(t) = s_2 \in$ I then $s_1 = s_2$;
2. for every closure function f and every ground term t, there is ground set s such that $f(t) = s \in$ I.

The first requirement states that f should be treated as a function; the second requirement insists on the totality of f, or its termination.

Definition: Given two sets of ground assertions I, J, define $I \sqsubseteq J$ iff for any $\texttt{f}(t) = s_1 \in I$ or $\texttt{f}(t) \supseteq s_1 \in I$ there exists $\texttt{f}(t) = s_2 \in J$ such that $s_1 \subseteq s_2$. When this is the case we say that *I is bounded by J*.

Definition: Let P be a stratified program. An interpretation M is a model of P denoted by $M \models P$ iff: for every ground instance, S :- $\mathrm{E}_1^g \ldots \mathrm{E}_k^g$, of an subset clause in P, if $\{\mathrm{E}_1^g \ldots \mathrm{E}_k^g\} \subseteq M$ then { S } is bounded by M.

---

† Strictly speaking, we should work with equivalence classes of terms and atoms, due to the equality theory of sets. However, we will talk of terms, instead of equivalence classes of terms, for simplicity of presentation.

Note: A program may be **inconsistent**, i.e., it may not have an Herbrand model. For example, the following program is inconsistent: $f(x) \supseteq \{x\}$, and $f(x) \supseteq f(\{x\})$. The function f does have a model in which $f(x)$ denotes an infinite set; however, infinite sets are not permitted in our Herbrand Universe.

Since we will construct the declarative semantics of a stratified program level by level, starting from level 1, in defining models at some level $j > 1$, all functions from levels $< j$ will have their models uniquely specified. Hence, all interpretations of clauses at some level j will contain the same assertions for every function from a level $< j$. For this reason we introduce the following notation, wherein we use *level*(A) to refer to the level of the head function symbol of atom A:

Definition: Let I be an interpretation. We define $I_k := \{ \text{A} : \text{A} \in I \text{ and } level(A) \leq k\}$.

Definition: Let I,J be two interpretations for a program P . We define

$$I \sqcup J := \{f(t) = s \cup s' \quad : \quad f(t) = s \in I, f(t) = s' \in J\}$$

$$I \sqcap J := \{f(t) = s \cap s' \quad : \quad f(t) = s \in I, f(t) = s' \in J\}$$

Definition: Let X be a set of interpretations for P. We define $\sqcup X$ and $\sqcap X$ as the natural generalization of the previous definition.

Proposition: Let X be a set of models for a P_j program such that for any $I \in X$ and $J \in X$, $I_{j-1} = J_{j-1}$, then $\sqcap X$ is also a model.

Definition: We define the *declarative semantics* of a consistent P_j program as:

for $j = 1$, $D(P_1) := \sqcap\{M : M \text{ models } P_1\}$, and

for $j > 1$, $D(P_j) := \sqcap\{M : M_{j-1} = D(P_{j-1}) \text{ and } M \text{ models } P_j\}$.

3.2. Operational Semantics

In preparation for the operational semantics for closure functions, we first define the *collect-all reduction* of a query expression G with respect to a subset program P starting from their respective flattened forms, in which the order of equalities reflects the innermost order of reducing expressions.

Definition : An extended goal is a pair of the form <G, T> where G is a goal-sequence , and T is a memo-table, i.e., a set of assertions of the form $\mathtt{f}(t) = u$, where `f` is a closure function, t is a ground term, but u may be non ground.

Definition: Given variants of subset assertions, $\mathtt{f}(t_1) \supseteq s_1$ `:-` B_1, ... $\mathtt{f}(t_n) \supseteq s_n$ `:-` B_n, in which variables have been suitably renamed, and given a query expression

$$\text{G} := \text{g}_1, \ldots, \text{g}_m,$$

where g_1 is $\mathtt{f}(t) = x$ and t is a ground term and x is a variable, we define the collect-all reduction relation between goals G , G' such that,

(a) if matching t with $t_1 \ldots t_n$ yields respectively the (finitely many) substitutions $\theta_{11}, \ldots, \theta_{1k_1}, \ldots, \theta_{n1}, \ldots, \theta_{nk_n}$, then

$$\text{G}' := (\text{B}_1\ \theta_{11}), \ldots, (\text{B}_1\ \theta_{1k_1}), \ldots, (\text{B}_n\ \theta_{n1}), \ldots, (\text{B}_n\ \theta_{nk_n}), (\text{g}_2, \ldots, \text{g}_m)\ \sigma,$$

where $\sigma := \{x \leftarrow \cup_{i=1,n} \cup_{j=1,k_i} (s_i \theta_{ij})\}$;

(b) if there are no matches between t and any t_i,

G' := (g_2, ..., g_n) σ, where $\sigma = \{x \leftarrow \phi\}$.

(The above reduction strategy can be shown to be sound and complete for *non-closure* functions.)

Definition : Given an extended goal <G, T>, let the first goal in G, g_1 be $\mathtt{f}(t) = v$. The relation <G, T> → <G', T'> is defined as follows.

(i) If there is no assertion of the form $\mathtt{f}(t) = w$ in T, then we reduce G → G' by a *collect-all reduction*, and T' := (T ∪ $\{\mathtt{f}(t) = v\}$) σ, where σ is the collect-all substitution for v in deriving G'.

(ii) If $\mathtt{f}(t) = w$ is in T, then G' := (G \ [g_1]) σ, and T' := T σ, where (G \ [g_1]) is the goal-sequence G with g_1 removed, and $\sigma := \{v \leftarrow w'\}$, where w' is the smallest solution to the equation $v = w$.

Definition: Given a program P and an extended goal G^E := <G, T>, we say that θ restricted to variables in G is the computed answer for P and G^E if there is a derivation

$$G^E = G^E_1 \rightarrow \; ... \; \rightarrow G^E_k = < [], T_k >$$

where θ_i is the substitution using in reducing G^E_i, $\theta = \theta_1 ... \theta_k$ and [] is the empty goal.

Soundness Theorem: Let $G^E := < f(t) = x, \phi >$ be an extended goal for a program P. Then the computed answer for G^E is correct for G.

Proof: See Appendix.

The following example shows that we do not have **completeness**: `f(X)` = `f({X})`. Notice that the query `f({1})` = Z does not have a computed answer—the procedure loops forever—but on the other hand it has a correct answer $\theta = \{Z \leftarrow \phi\}$.

3.3 General Stratified Programs

The **simple stratified language** defined in section 3.1 permits the definition of one closure function *directly* in terms of another closure function at the same level. However, the **general stratified language** defined below permits the definition of one closure function in terms of another closure function at the same level using *subset-monotonic* functions.

Definition: A set-valued function f is subset-monotonic in its i^{th} argument iff $s_1 \subseteq s_2 \Rightarrow f(\ldots, s_1, \ldots) \subseteq f(\ldots, s_2, \ldots)$, where the i^{th} argument is the one shown and all other argument positions remain unchanged in $f(\ldots, s_1, \ldots)$ and $f(\ldots, s_2, \ldots)$.

For *general stratified* programs, the assertions are as in the simple case, plus the following assertion scheme.

$$f(terms) \supseteq g(\ldots, p(lexpr), \ldots)$$

where $f$and p are closure functions at some level j, any other occurrence of a function in the assertion is of level less than j. The function g is subset-monotonic in the argument where g appears. That is, a closure function at level j is either directly defined in terms of other closure functions at the same level or defined in terms of a subset-monotonic function.

It is straightforward to show that the presence of subset-monotonic functions does not call for any alteration of the declarative semantics. The operational semantics, however, must be modified to incorporate the *re-do* mechanism. The correctness of this procedure is also easy to establish, but space limitations precludes a complete treatment here.

4. Relation with Semantics of Negation-as-Failure

We now show another legitimate way to characterize the semantics of general stratified programs of section 3. Given a general stratified program P we will translate it to a *stratified normal program*, $extension(P')$, and then we will take the *stratified model* of $extension(P')$ as the semantics of our program. We prove that the stratified model of $extension(P')$ corresponds to $D(P)$ of section 3.

4.1. Translation to Normal Programs

We motivate our approach with the `reach` example of section 2.2. We henceforth refer to these assertions collectively as `Reach`.

$\texttt{closure reach}$
$\texttt{reach}(\{\texttt{X}\backslash\_\}) \supseteq \{\texttt{X}\}$
$\texttt{reach}(\{\texttt{X}\backslash\_\}) \supseteq \texttt{reach}(\texttt{edge}(\texttt{X}))$
$\texttt{edge}(1) \supseteq \{2\}$
$\texttt{edge}(2) \supseteq \{3\}$

In flattened form, all assertions remain the same, except the second assertion, which would be as follows:

$\texttt{reach}(\{\texttt{X}\backslash\_\}) \supseteq \texttt{S :- edge}(\texttt{X}) = \texttt{X}_1,\ \texttt{reach}(\texttt{X}_1) = \texttt{S}$

We convert the flattened form into *normal program clauses* [L87], as follows:

$\texttt{reach}_{\supseteq}(\{\texttt{X}\backslash\_\}, \{\texttt{X}\})$
$\texttt{reach}_{\supseteq}(\{\texttt{X}\backslash\_\}, \texttt{S}) \texttt{ :- } \texttt{edge}_{=}(\texttt{X, X}_1), \texttt{reach}_{=}(\texttt{X}_1, \texttt{S})$
$\texttt{edge}_{\supseteq}(1,\ \{2\})$
$\texttt{edge}_{\supseteq}(2,\ \{3\})$

In our intended interpretation, $\texttt{reach}_{=}(\{x\},\ \{y\backslash\_\})$ is true iff y belongs to the transitive closure of $\texttt{edge}_{\supseteq}(x, \{y\})$. In order to try to capture this intended interpretation, we extend the program P by adding the following set of normal clauses.

$\texttt{reach}_{\supseteq}(\texttt{Z}, \phi)$
$\texttt{reach}_{\supseteq}(\texttt{Z}, \{\texttt{X}\backslash\texttt{Y}\}) \texttt{ :- } \texttt{reach}_{\supseteq}(\texttt{Z}, \{\texttt{X}\backslash\_\}),\ \texttt{reach}_{\supseteq}(\texttt{Z}, \texttt{Y})$
$\texttt{reach}_{=}(\texttt{Z, S}) \texttt{ :- } \texttt{reach}_{\supseteq}(\texttt{Z, S}),\ \neg\, \texttt{reach}_{\not\supseteq}(\texttt{Z}, \texttt{S})$
$\texttt{reach}_{\not\supseteq}(\texttt{Z, S}) \texttt{ :- } \texttt{reach}_{\supseteq}(\texttt{Z}, \{\texttt{Y}\}),\ \texttt{Y} \notin \texttt{S}$

A similar collection of assertions is also added for `edge`. We denote this extended program by $extension(P)$. Note that, in this case, $extension(P)$ is a non-stratified normal program and that the Stable Model Semantics [GL88] as well as the Well Founded Semantics [VRS91] define our intended model.

Unfortunately, the above extension fails to capture our intended model if we add even the assertion $\texttt{edge(3)} \supseteq \{2\}$ to our program P. Observe that now our intended model M for $extension(P)$ is not a stable set since $\texttt{reach}_{=}(1, \{1, 2\})$ is not in the minimal Herbrand model of the Gelfond-Lifschitz Transformation of P with respect to M [GL88]. Moreover $extension(P)$ has no well founded total model. Essentially the problem arises due to the loop introduced into the graph by the definition of the predicate $\texttt{edge}_{\supseteq}$. To circumvent the above problem, we will work with $extension(P')$ which is as $extension(P)$ but instead of having the assertion

$\texttt{reach}_{\supseteq}(\{\texttt{X}\backslash\_\}, \texttt{S}) \texttt{ :- } \texttt{edge}_{=}(\texttt{X, X}_1), \texttt{reach}_{=}(\texttt{X}_1, \texttt{S})$

we have

$$\mathtt{reach}_{\supseteq}(\{\mathtt{X}\backslash\_\},\mathtt{S})\ \mathtt{:-}\ \ \mathtt{edge}_{=}(\mathtt{X},\ \mathtt{X}_1),\ \mathtt{reach}_{\supseteq}(\mathtt{X}_1,\mathtt{S})$$

Note that in this case *extension*(P') is stratified and its stratified semantics clearly corresponds to our intended model.

We now formalize the notion of program extension. To simplify the exposition, we assume for the rest of the section that a program is of the form P_2 and that we know the intended interpretation for P_1 is known. Henceforth, P will stand a program of this kind.

Definition: Given a program P, we define *head*(P) as the set of head symbols of P, i.e., the head symbols on the literals on the left-hand sides of subset assertions.

Definition: Given a function symbol `f`, we define *ext*(`f`) as the following set of assertions:

$$\begin{array}{lll}
\mathtt{f}_{\supseteq}(\mathtt{Z},\phi) & & \\
\mathtt{f}_{=}(\mathtt{Z},\mathtt{S}) & \mathtt{:-}\ \mathtt{f}_{\supseteq}(\mathtt{Z},\mathtt{S}), & \neg\ \mathtt{f}_{\not\supseteq}(\mathtt{Z},\mathtt{S}) \\
\mathtt{f}_{\supseteq}(\mathtt{Z},\{\mathtt{X}\backslash\mathtt{Y}\}) & \mathtt{:-}\ \mathtt{f}_{\supseteq}(\mathtt{Z},\{\mathtt{X}\backslash\_\}), & \mathtt{f}_{\supseteq}(\mathtt{Z},\mathtt{Y}) \\
\mathtt{f}_{\not\supseteq}(\mathtt{Z},\mathtt{S}) & \mathtt{:-}\ \mathtt{f}_{\supseteq}(\mathtt{Z},\{\mathtt{Y}\backslash\_\}), & \mathtt{Y}\notin\mathtt{S}.
\end{array}$$

Definition: Given a program P, we define $ext(P) := \bigcup_{f\in\ head(P)} ext(f)$

Definition: Given P, we define $extension(P) := P \cup ext(P)$.

4.2 Stratified Semantics

We assume that the reader is familiar with the definition of the Stratified Semantics [ABW88] (abbreviated here as SS). We assume that the SS has been extended to deal with our set constructor.

Definition: Given a program P, we define P' to be the same as P except that we replace each assertion of the form

$$E_1\ \mathtt{:-}\ E_2,\ldots,E_k,\ldots,E_n$$

by the assertion

$$E_1 \colon - E_2,\ldots,E_k^*,\ldots,E_n$$

where E_1 is of the form $f(t_1) \supseteq x_1$, E_k is of the form $g(t_k) = x_k$, E_k^* is of the form $g(t_k) \supseteq x_k$ and f and g are (not necessarily different) closure functions at the same level. Note that when P is simple-stratified we have $k = n$.

Proposition: For any program P, *extension*(P') is stratified.

This result simplifies the study of some properties of *extension*(P'), a major one of which is the following:

Proposition: For any program P, *extension*(P') has a stratified model.

Definition: Given a program P, we define Ax_P to be the following set of axioms, one set for each closure function $f \in P$.

$$f(x) \supseteq y \rightarrow \exists! z(f(x) = z \wedge z \supseteq y)$$

$$f(x) = y \rightarrow f(x) \supseteq y$$

$$(f(x) \supseteq y \wedge y \supseteq z) \rightarrow f(x) \supseteq z$$

For *general stratified programs*, we would also include the axiom of subset-monotonicity for the relevant functions.

Proposition: Given a program P, $P \cup Ax_P$ is logically equivalent to $P' \cup Ax_P$.

We say that a subset program is consistent if it is consistent in the sense of section 3.1, and we say that its translation is consistent if it is consistent as a normal program.

Proposition: If P is a consistent program then Ax_P holds in $\mathrm{SS}(extension(P'))$.

Proposition: Any consistent program P satisfies the following properties:

1. $f(t) = s \in D(P)$ implies $f(t) \supseteq s \in \mathrm{SS}(extension(P'))$.
2. $f(t) = s \in \mathrm{SS}(extension(P'))$ implies $f(t) = s_1 \in \mathrm{D(P)}$, for some $\mathrm{s}_1 \supseteq s$.

From the above two propositions, we obtain our main result:

Theorem: For any consistent program P, $f(t) = s \in D(P)$ if and only if $f(t) = s \in \mathrm{SS}(extension(P'))$.

It is interesting to observe that a program P could be inconsistent but $extension(P')$ be consistent. For instance let P be:

$\mathtt{f(X)} \supseteq \{\mathtt{X}\}$
$\mathtt{f(X)} \supseteq \mathtt{f}(\{\mathtt{X}\})$

As mentioned in section 3, P is inconsistent but $extension(P')$ is consistent and moreover it possesses a stratified model where f is a partial function.

4.3. Well-Founded Semantics

Let us return to analyze the original extension *extension* for the program `Reach`, but taking the following definition of the predicate `edge`:

$\mathtt{edge(1)} \supseteq \{1\}$

$\mathtt{edge(2)} \supseteq \{3\}$

$\mathtt{edge(3)} \supseteq \{2\}$

(We ask the reader to translate these assertions and others in this subsection into their normal program clause equivalents.) We remark that the stable semantics fails to have a stable model for this program. Now, the well founded semantics gives a partial model. We argue that the most sensible extended model, provided that `reach` is a function, is in fact the intended model. First note that the well-founded partial model agrees in its true/false assignments with the intended model. The undefined values for `reach(2)` are:

$\mathtt{reach(2)} \supseteq \{3\}, \mathtt{reach(2)} \supseteq \{2,3\}, \mathtt{reach(2)} = \{2\},$

$\mathtt{reach(2)} = \{2,3\}, \mathtt{reach(2)} \not\subseteq \{2\}, \mathtt{reach(2)} \not\subseteq \{1,2\}$

Two interesting points where the well founded model agrees with the intended model are:

$\mathtt{reach(1)}=\{1\}$ and $\neg\mathtt{reach(1)} = \{1,2,3\}$.

Now we analyze the possible extensions of this partial model to get a total model:

Case 1: Suppose `reach(2)` $= \{2\}$. By the symmetry of the syntax we should assume also that `reach(3)` $= \{3\}$. We get then that this model is not supported.

Case 2: Suppose $\neg\texttt{reach}(2) = \{2\}$.

Case 2a: Suppose also $\neg\texttt{reach}(2) = \{2, 3\}$. Here `reach(2)= s` is false for any ground term `s`, meaning that $\texttt{reach}_=$ is not a total function.

Case 2b: Suppose on the other hand that $\texttt{reach}(2) = \{2, 3\}$. This is our intended model. In fact in the well-founded model of $\mathit{extension}(\texttt{Reach}) \cup \{\texttt{reach}(2) = \{2, 3\}\}$, we have that $\neg\texttt{reach}(2) = \{2\}$ is true.

The previous analysis shows that the only two sensible extended models are the intended model and the model where $\texttt{reach}_=$ behaves as a partial function. This suggests that an extension of the well founded semantics to allow function definitions should agree with our intended model.

6. Related Work and Further Extensions

A variety of approaches to sets have been proposed for logic programming languages [JP87, BN*87, CW92, DOPR91, J92, K90]. The goal of this paper was not to resolve the relative merits of various approaches; rather, our goal was to clarify the connections between subset assertions and semantics of negation, especially stratified semantics and well-founded semantics. Nevertheless, we claim that our support for finite sets marks an significant advance over all other approaches described in the literature, especially in our ability to define compactly and elegantly the class of flow analysis problems illustrated in section 2.3.

We note that our proposed language described in this paper is essentially a functional language, although its extension to accomodate logic programming features are discussed in [JP89, J91]. The traditional need for memo tables in implementations of functional languages has been to avoid unnecessary computation, by detecting when a function call with identical arguments has been earlier computed. This idea was first put forward in [M68], by Michie, who called such functions *memo functions*. This idea was later elaborated in [FWW76, KS81, H85]. In all these cases, memo tables are proposed in order to gain performance improvements; the semantics of the underlying language is unaffected by memoization. In this paper we discuss a paradigm of functional programming where memo tables are needed for semantic reasons.

In comparison with logic programming languages supporting sets, especially LDL [BN*87], an interesting aspect of our work is that our proposed construct is amenable to a simple least model semantics, by taking intersection of sets, as defined by $\sqcap$. Furthermore, we have a more liberal definition of stratification. These in turn are due to the fact that our language is essentially a functional language. Further comparisons with LDL are in the paper [JP89]. Our set constructors resemble those of [DOPR91], but can be traced back to those in [JP87, JP89]. Quantifiers over sets are discussed in [K90, DOPR91], a feature that is not explicitly found in our language. Finally, our translation of subset assertions into normal program clauses [L87] in fact suggests a way of providing the semantics of the *setof* construct in logic programming languages.

We showed that memoization and re-do forms a natural (top-down) evaluation strategy for subset assertions. The use of re-do seems related to the idea of *reexecution* discussed in [LV92]. It is noteworthy that both operational mechanisms are motivated by a common application area, i.e., program analysis. To appreciate the full capability of memoization and re-do, consider the following formulation of the shortest distance between a vertex pair in a graph, which is assumed here to be represented as an adjacency set.

```
closure costs, short

short(X,Y) = if eq(X,Y) then 0 else min(costs(edgesfrom(X), Y))
costs({[Z,D]\_},Y) ⊇ if eq(Z,Y) then D else {D+short(Z,Y)}
```

In this formulation we have assumed the availability of equational assertions and the usual if-then-else conditional. The function `edgesfrom(X)` (not shown) defines the adjacency set for the graph, by returning the set of pairs `[Z,D]` where `Z` is an immediate neighbor of `X` and the distance from `X` to `Z` is `D`. The function `min` (not shown) returns the smallest number in a nonempty set, and it returns the largest possible distance ("infinity") if the set is empty. Note that `short` is defined as a closure function even though it is not set-valued. However, the above program satisfies the requirements for a well-defined semantics because the function `min` is *monotonic* in the following sense: $s_1 \subseteq s_2 \Rightarrow \mathtt{min}(s_1) \geq \mathtt{min}(s_2)$. Thus, if the graph is cyclic and a look-up operation is performed, the initial look-up value would be "infinity". As the computation proceeds, `min` progressively returns smaller values until a fixed-point is reached. As an optimization, by further annotating that `min` distributes over union, it is possible to avoid forming any intermediate set, thereby achieving in a purely declarative way the kind of efficiency that one might achieve through traditional dynamic programming. Thus the natural generalization of the paradigm would be to any lattice, so that there is an appropriate "least" element (i.e., the initial look-up value), with the requirement that all closure functions should be defined in terms of one another using monotonic functions with respect to the appropriate partial orderings.

Acknowledgments

The implementation of subset assertions was carried out by Kyonghee Moon, a doctoral student at SUNY-Buffalo, and all program examples in this paper were tested out using this implementation. The XOLDT system developed by D.S. Warren was valuable in testing out the translated normal programs. This research was supported in part by grant DCR 9004357 from the National Science Foundation.

References

[ABW88] Apt,K., Blair, H., and Walker, A. "Toward a theory of declarative knowledge," *Proc. of the Workshop on Foundations of Deductive Database and Logic Programming* (1988) p. 89-148.

[AU77] A. Aho and J.D. Ullman, "Principles of Compiler Design," Addison-Wesley, 1977.

[BN*87] C. Beeri, S. Naqvi, et al, "Sets and Negation in a Logic Database Language (LDL1)," *Proc. 6th ACM PODS*, pp. 21-37, 1987.

[CW92] W. Chen and D.S. Warren, "An Intensional Logic for (Multi-Arity) Set Abstraction," *Proc. 8th ICLP*, pp. 97-110, Paris, June 1991.

[DOPR91] Dovier, A., Omodeo, E. G., Pontelli, E., and Rossi, G. "{log}: A Logic Programming Language with Finite Sets," *Proc. 8th ICLP*, pp. 111–124, Paris, June 1991.

[FWW76] D. Friedman, D. Wise, and M. Wand, "Recursive Programming Through Table Look-up," *Proc. Symp. on Symbolic and Algebraic Computation*, pp. 85-89, 1976.

[GL88] M. Gelfond and V. Lifschitz, "The Stable Model Semantics for Logic Programming," *Proc. 5th ICLP*, pp. 1070-1080, Seattle, August 1988.

[H85] R.J.M. Hughes, "Lazy Memo Functions," *Proc. Second FPCA*, Nancy, Sept. 1985, pp. 129-146.

[H92] Heintze, N., "Practical Aspects of Set Based Analysis," *Proc. JICSLP 92*, pp. 735-749, Washington DC, 1992.

[J91] B. Jayaraman, "the SuRE Programming Framework," Dept. of Computer Science TR 91-011, SUNY-Buffalo, July 1991.

[J92] B. Jayaraman, "Implementation of Subset-Equational Programs," *JLP*, 11(4):299-324.

[JP87] Jayaraman, B. and Plaisted, D. A., *Proc. Third Intl. FPCA*, Portland, OR, September 1987, pp. 194-210, Springer-Verlag.

[JP89] Jayaraman, B. and Plaisted, D. A., "Programming with Equations, Subsets, and Relations," *Proc. NACLP 89*, pp. 1051-1068, Cleveland, October 1989.

[K90] Kuper, G. M., "Logic Programming with Sets," *JCSS*, 41(1):44-64, 1990.

[KS81] R.M. Keller and M.R. Sleep, "Applicative Caching: Programmer Control of Object Sharing and Lifetime in Distributed Implementations of Functional Languages," *Proc. First FPCA*, Portsmouth, NH, Oct. 1981, pp. 131-140.

[LV92] B. Le Charlier, P. van Hentenryck, "Reexecution in Abstract Interpretation of Prolog," *Proc. JICSLP 92*, pp. 750-764, Washington DC, November 1992.

[L87] J.W. Lloyd, "Foundations of Logic Programming," (2 ed.) Springer-Verlag, 1987.

[M68] D. Michie, "'Memo' Functions and Machine Learning," *Nature*, 218:19-22, 1968.

[VRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf, "The Well-Founded Semantics for General Logic Programs," *JACM*, 38(3):620-650.

[W92] D.S. Warren, "Memoing for Logic Progams," *CACM*, 35(3):93-111, 1992.

Appendix: Soundness Proof for Simple-Stratified Programs

We establish the soundness for simple-stratified programs by induction on the number of levels. For this proof, it suffices to treat level i assertions as having the same structure as level 1 assertions, shown below, because, in the reduction of a level i expression, all expressions of lower levels than i would have been reduced to *terms* by the operational semantics:

$$f(terms) \supseteq g(term)$$
$$f(terms) \supseteq set$$

where f and g are closure functions. The flattened form of such a program clause $f(t_1) \supseteq g(t_2)$ (where f and g not necessarily distinct) will be:

$$f(t_1) \supseteq x \text{ :- } g(t_2) = x$$

In addition, of course, a flattened form might be a