Subset-logic Programming: Application and Implementation

Bharat Jayaraman
Anil Nair

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

Abstract

Subset-logic programming is a paradigm of programming with subset and equality assertions. Computationally, equality and subset assertions are treated as rewrite rules, where the matching operation is a restricted form of associative-commutative (a-c) matching. The multiple matching substitutions 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. We present a language called SEL to illustrate the approach. We also show that WAM-like instructions can be used to compile restricted a-c matching. An important property of a SEL function is whether or not it 'distributes over union' in a particular argument. If it does, we can avoid checking for duplicates in this argument, and also avoid constructing the set corresponding to this argument.

1. Introduction

The term 'logic programming' is often taken to be synonymous with predicate-logic programming, owing to the latter's simple semantics [5] and the success of Prolog. In recent years, other forms of logic programming have been proposed, most notably equational-logic [9] and constraint-logic programming [3]. We contribute another such approach in this paper, called subset-logic programming. The goal of our work is to provide a rigorous basis for programming with sets. Existing approaches, such as the 'setof' constructs of Prolog systems [8], are not supported by an underlying logic although they are very useful in practice. In our proposed approach, a program is a collection of two kinds of assertions:

(i) *f(terms)* $=$ *expression*

(ii) *f(terms)* $\supseteq$ *expression*

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

? *expr*

where *expr* is a ground expression. The meaning of this query is the ground term g such that $expr = g$ is a logical consequence of the program assertions supplemented with those equality assertions that are derived from closed-world considerations.

The language framework for conveying these ideas is called SEL, for Set-Equation Language. The data objects in SEL, called terms, are the finite objects built up from atoms and data-constructors. (There are no infinite or higher-order objects in the current version of SEL.) Terms are distinguished from more general expressions, which may also contain function applications. Apart from the usual data-constructors of Prolog, we also permit the associative-commutative (a-c) constructor $\cup$, for set union. The $\cup$ constructor is our means of defining sets.

Computation with these assertions is a process of 'replacing equals by equals'. Both equality and subset assertions are oriented left-to-right for rewriting. All constructors and user-defined functions are *strict* in all arguments; thus, nested function applications are performed innermost-first. Because arguments to functions are ground terms, function application requires one-way matching, rather than unification. The matching operation is associative-commutative (a-c) matching [10], because of the presence of the $\cup$ constructor. In this paper, we restrict the use of $\cup$ in program assertions in a manner that supports clear programming as well as efficient implementation. The associated matching algorithm is referred to as restricted a-c matching.

SEL is essentially a functional programming language in which sets are ‘first class’ objects, i.e., not simulated by lists. Its benefits for functional and logic programming are: (i) many operations over sets can be stated non-recursively, thanks to the implicit iteration over sets provided by restricted a-c matching; (ii) nondeterministic search can be specified without the use of ‘cuts’; (iii) efficient (non-backtrackable) execution is possible with equations; and (iv) checks for duplicate elements in argument sets and formation of intermediate sets can be avoided when operations using these sets ‘distribute over union’ (discussed in section 2).

SEL does *not* support unification or backward reasoning. We believe these capabilities are already well-supported in predicate- and constraint-logic programming. A unified language with both capabilities can be designed, but this issue is beyond the scope of this paper.

In order to demonstrate the practicality of our approach, we also present in this paper the implementation of SEL programs. Our implementation model is essentially a stack-heap model based on structure copying. It turns out that ‘WAM’-like instructions [13] are very appropriate for the compilation of restricted a-c matching. Because we employ one-way matching, we can identify at compile-time the ‘read’ and ‘write’ modes of WAM’s ‘get’ instruction. Another interesting contrast from Prolog implementations is that backtracking in a SEL implementation could occur both on success as well as failure. The former occurs because multiple branch points could arise in the invocation of a single subset assertion—due to branching in a-c matching—and the successful completion of one such branch requires backtracking to repeat the same right-side, but using a different matching substitution.

We described the formal operational semantics of subset-logic programming in our earlier paper [4], and are in the process of completing a formal declarative semantics. The presentation in this paper is therefore informal; our main objective is to present the basic intuition behind subset-logic programming, to describe restricted a-c matching, and also to demonstrate that it can be implemented efficiently using WAM-like instructions. The rest of this paper is organized as follows: section 2 informally presents the features of SEL, restricted a-c matching, and examples; section 3 describes an abstract machine for SEL: its execution model, instruction set, and the compiled code for a typical program; and section 4 presents conclusions, possible extensions, and further comments on related work.

2. Subset-logic Programming

We first specify the syntactic structure of *term* and *expression.*

$$\begin{aligned}
&\mathit{term} ::= \mathit{atom} \mid \mathit{variable} \mid \{\ \} \mid \{\mathit{term}\} \mid \mathit{term} \cup \mathit{term} \mid \\
&\qquad\quad \mathit{constructor}(\mathit{terms}) \\
&\mathit{terms} ::= \mathit{term} \mid \mathit{term}\ ,\ \mathit{terms} \\
&\mathit{expr} ::= \mathit{term} \mid \{\mathit{expr}\} \mid \mathit{expr} \cup \mathit{expr} \mid \mathit{constructor}(\mathit{exprs}) \mid \\
&\qquad\quad \mathit{function}(\mathit{exprs}) \mid \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \\
&\mathit{exprs} ::= \mathit{expr} \mid \mathit{expr}\ ,\ \mathit{exprs}
\end{aligned}$$

We use the [. . .] notation for writing lists, as in Prolog, and also the notation [h | t] to refer to a non-empty list, with head h and tail t. Similarly, we use the $\{\ldots\}$ notation for sets, e.g. $\{1, 2, 3\}$, and also use $\{x \mid t\}$ to refer a non-empty set, one of whose elements is x and the remainder of the set is t. Thus, $\{x \mid t\} \equiv \{x\} \cup t$. The set $\{1, 2, 3\}$ may be represented as $\{1\} \cup \{2\} \cup \{3\} \cup \{\ \}$. Other permutations, such as $\{2\} \cup \{1\} \cup \{3\} \cup \{\ \}$, $\{1\} \cup \{3\} \cup \{2\} \cup \{\ \}$, etc., represent the same set.

2.1 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 $t_1\theta =_{ac} t_2$? This problem was first posed by Plotkin [10] and has since been studied quite extensively in the literature, and recently been shown to be NP-complete [1]. In this paper, we propose a restriction that preserves programming convenience and makes possible efficient compilation. We disallow *explicit* use of the $\cup$ constructor in SEL assertions. Instead, we permit arbitrary combinations of patterns of the form

$\{term \mid term\}$.

Basically, this restriction permits iteration over the elements of a set, rather than iteration over the subsets of a set. While some expressive convenience is sacrificed by this restriction, most practical cases are unaffected. This restriction turns out to be very important for compilability of SEL programs. We refer to the associated matching operation as restricted a-c matching.

Note that the equality $=_{ac}$ is based only the associative and commutative properties, but not the idempotent property. Thus, for example, matching $\{\mathtt{h} \mid \mathtt{t}\}$ with $\{\mathtt{1}, \mathtt{2}, \mathtt{3}\}$ cannot yield the matching substitution $\{\mathtt{h} \leftarrow \mathtt{1}, \mathtt{t} \leftarrow \{\mathtt{1}, \mathtt{2}, \mathtt{3}\}\}$. The reason for disallowing the idempotent property during matching will be clear when we consider recursive SEL assertions. Because a singleton set $\{\mathtt{1}\}$ is represented internally as $\{\mathtt{1}\} \cup \{\ \}$, it can match $\{\mathtt{h} \mid \mathtt{t}\}$ yielding $\{\mathtt{h} \leftarrow \mathtt{1}, \mathtt{t} \leftarrow \{\ \}\}$; thus the identity property is not explicitly required during matching.

Below we present a Prolog program to specify more precisely the behavior of the restricted matching algorithm. (However, we do not literally follow this recursive procedure in our proposed implementation of a-c matching in section 3.) The first argument of `match` is a possibly non-ground term, representing the head of an assertion, and the second argument is a ground term, representing the arguments of a function call. In case a match is possible, the variables in the first input argument are instantiated appropriately. Multiple matches are produced one at a time. For simplicity, only lists and sets are considered; other constructors can be treated similarly.

```
match(A, A) :-
        atomic(A), !.
match({ }, { }).
match(V, Arg) :-
        var(V), !,
        V = Arg.
match([T1 | T2], [Arg1 | Arg2]) :-
```

```
          match(T1, Arg1),
          match(T2, Arg2).
match({Elem1 | Set1}, ArgSet) :-
          generate(ArgSet, Elem2, Set2),
          match(Elem1, Elem2),
          match(Set1, Set2).
generate({Elem | Set}, Elem, Set).
generate({Elem | Set}, Elem2, {Elem | Set2}) :-
          generate(Set, Elem2, Set2).
```

2.2 Program Assertions

As mentioned in the introduction, program assertions are either of the form

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

We require that every variable on the right-side of an equality or subset assertion must be present on its left-side. There are no free variables in SEL. We informally explain the operational semantics of these assertions; a more formal account is given in our earlier paper [4] in terms of rewrite rules.

For example, when matching an expression $\mathtt{distr}(10, \{1, 2, 3\})$ with the left-side of a subset assertion

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

all three matches are considered, namely, $\{\mathtt{x} \leftarrow 10, \mathtt{h} \leftarrow 1, \mathtt{t} \leftarrow \{2, 3\}\}$, $\{\mathtt{x} \leftarrow 10, \mathtt{h} \leftarrow 2, \mathtt{t} \leftarrow \{1, 3\}\}$, and $\{\mathtt{x} \leftarrow 10, \mathtt{h} \leftarrow 3, \mathtt{t} \leftarrow \{1, 2\}\}$. 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 $\mathtt{distr}(10, \{1, 2, 3\})$. Thus, the value returned in this case would be $\{[10|1], [10|2], [10|3]\}$. In general, duplicate elements are eliminated while taking this union—we mention in section 2.4 when we can avoid checking for duplicates and also avoid constructing this set. If multiple subset assertions match a call, their respective right-sides are similarly reduced, and the union of all such results is taken as the result of the call. Because the union operation is strict, it will not terminate if any of these reductions does not terminate. However, because of the closed-world assumption, if any of one these reductions terminates with a non-term expression, its result can be assumed to be { } for the purpose of the union. This marks another difference between the language described here and that of our earlier description [4].

Unlike subset assertions, when computing with equality assertions, only one of the potentially many a-c matches is considered in reducing the matching assertion, because we assume the result of rewriting is independent of which particular match is considered. For example, when matching an expression `size({1,2,3})` with the left-side of an assertion

```
size({h | t})  =  1 + size(t)
```

any one of the three matches for `h` and `t` may be taken, and the others ignored. It is left to the programmer to ensure that the result of rewriting is independent of the particular match considered—in our earlier paper [4], we mentioned methods of proving confluence for equational programs with a-c matching. An example of an assertion that violates this property is: `set2list({h | t}) = [h | set2list(t)]`.

Finally, we define the conditional expression as follows:

`if true then e1 else e2 = e1`, and

`if` x `then e1 else e2 = e2`, if x terminates and $x \neq$ `true`

That is, the conditional expression implements a form of *negation by failure* [2].

2.3 Examples of SEL Programs

List Operations:

```
reverse([ ]) = [ ]
reverse([h | t]) = append(reverse(t), [h])

append([ ], y) = y
append([h | t], y) = [h | append(t, y)]
```

First-order functional programming with lists can be carried out in the usual way with equations, as the above example is meant to suggest.

Set Operations:

```
crossproduct({x | _ }, {y | _ }) ⊇ {[x | y]}
intersect({x | _ }, {x | _ }) ⊇ {x}
union(s1, s2) ⊇ s1
union(s1, s2) ⊇ s2
```

In `crossproduct` and `intersect`, no assertions are needed for the cases when the argument sets are empty; the result is the empty set in these cases, by the closed-world assumption. The anonymous variable _ is similar to that of Prolog. An important difference here, however, is that considerable space and time can be saved by not constructing the remainder of the set.

Permutations:

```
perms({ }) = {[ ]}
perms({x | t}) ⊇ distr(x, perms(t))
distr(x, {y | _ }) ⊇ {[x | y]}
```

The function `perms` takes a set of elements as input and produces as output the set of 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 its second-argument set.

Four Queens Problem:

```
queens(col, safeset) =
            if eq(col, 5) then safeset
            else placequeen(col, {1, 2, 3, 4}, safeset)
placequeen(col, {row | _ }, safeset) ⊇
            if safe([col | row], safeset)
            then queens(col + 1, {[col | row] | safeset})
            else { }
safe([c1 | r1], { }) = true
safe([c1 | r1], {[c2 | r2] | s}) =
            (r1 ≠ r2) and (abs(c1 − c2) ≠ abs(r1 − r2))
            and safe([c1 | r1], s)
?queens(1, { })
```

The above example illustrates how a search may be specified. The algorithm places a queen on each successive column, beginning from column 1, as long as each new queen placed is safe with respect to all queens in the preceding columns. A solution is found if a queen can be thus be placed on all columns. The second argument to `placequeen`, viz., the set $\{\texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}\}$, enumerates the row positions in each column. If a particular row-column position is not safe, `placequeen` returns the empty set $\{\ \}$, thereby pruning this line of search. The function `safe` specifies the safety condition—we assume that SEL has the usual complement of arithmetic operations.

2.4 Remarks

1. Note that the set operations `crossproduct`, etc., are all stated non-recursively. It is possible to compile these definitions so that no recursive calls occur even during matching.

2. The permutations example also illustrates why the idempotence property is not used during matching. If it were used, a set $\{\texttt{1}, \texttt{2}, \texttt{3}\}$, for example, could match $\{\texttt{x} \mid \texttt{t}\}$ in the second assertion for `perms` yielding $\{\texttt{x} \leftarrow \texttt{1}, \texttt{t} \leftarrow \{\texttt{1}, \texttt{2}, \texttt{3}\}\}$ as one of the matches. When the right-side of the assertion is reduced with this substitution, an infinite recursion would result.

3. In the permutations example, the n different matches of an n-element set with the pattern $\{\texttt{x} \mid \texttt{t}\}$ can be constructed, in a sequential implementation, in O(n) space rather than O(n^2) space. Each new remainder set for `t` can be constructed by destructively modifying the preceding value of `t`. In general, it is possible to construct these n remainder sets in O(n) space if `t` is not being returned (either directly or indirectly) as part of the function's result. Detecting this case necessitates data-flow analysis of the program assertions.

4. We say that an operation `f` *distributes over union in its i-th argument* iff

$$\texttt{f}(\ldots, \texttt{x} \cup \texttt{y}, \ldots) = \texttt{f}(\ldots, \texttt{x}, \ldots) \cup \texttt{f}(\ldots, \texttt{y}, \ldots)$$

where the *i*-th argument of `f` is the one shown above. Functions that compute some aggregate property of a set, e.g. `size` and `perms`, do not distribute over union. Functions, such as `distr` and `intersect`, that are defined in terms of the elements of the set do distribute over union. There are two benefits of knowing that a function distributes over union in a particular argument:

(i) We can avoid checking for duplicate elements in this argument; the function is simply applied to the singleton-sets that make up the argument set, and the individual results propagated. Because argument sets are usually free from duplicates, this can lead to substantial savings in execution time.

(ii) When several such functions are composed together, we effectively avoid constructing intermediate sets, thus saving space as well. This optimization is similar to the avoidance of constructing intermediate lists when composing a series of 'map' functions in functional languages.

At this stage of its development, we assume that a SEL programmer specifies, through suitable 'mode' declarations, in which arguments a function distributes over union.

3. Implementation

We now present here the salient aspects of an abstract machine for implementing SEL. This abstract machine is very similar to the WAM, being based on a stack-heap model with structure-copying. We therefore concentrate on the differences in this presentation. We assume that the reader has some familiarity with the WAM implementation of Prolog [13].

The basic approach is as follows: At compile-time, we flatten all expressions in accordance with innermost-first semantics, so that the arguments of all function calls are terms. Temporary variables are introduced as necessary. We illustrate by showing the flattened form of `perms` below.

$$\begin{array}{l}\texttt{perms}(\{\ \}) = \{[\ ]\} \\ \texttt{perms}(\{\texttt{x} \mid \texttt{t}\}) \supseteq \texttt{v1} \ \ :-\ \ \texttt{perms}(\texttt{t}) \supseteq \texttt{v2},\ \ \texttt{distr}(\texttt{x},\texttt{v2}) = \texttt{v1}\end{array}$$

Note that the operation `distr` distributes over union (in its second argument); this is distinguished in the flattened code by using $\supseteq$ rather $=$ in the goal containing the call to `perms`. Equality and subset assertions can be assumed to be mutually exclusive, i.e., an equality and a subset assertion cannot both match a given call. Furthermore, equality assertions can be assumed to be mutually exclusive among themselves; in case of overlap, the choice is arbitrary. Within each class, the assertions are indexed on their first argument, as in the WAM. We try all equality assertions first, followed by subset assertions.

The main data areas are: (i) the static *code area*, (ii) the *control stack*, and (iii) the *heap*. There is *no* need for a *trail* stack, because the matching is strictly one-way; trying alternative branches during a-c matching requires changes only to local variables. In addition to these areas, a push-down list is maintained in order to traverse nested structures during matching—similar to that needed for unification.

As in the WAM, the control stack is made up *environments* and *choice-points*. Environment trimming and last-call optimization are applicable to both equality assertions and subset assertions, although in the latter case these optimizations are applicable only to the last match among the multiple a-c matches. The heap stores lists, structures, and sets. Unlike the WAM, we do not need to identify *global* variables, because all returned values must be ground. In other words, all variables can be allocated on the control stack.

3.1 Execution Model

A function defined exclusively by equality assertions is invoked by a `call` instruction. An *environment* record is created on the *control stack* for this call if the matching assertion has *permanent* variables, as in the WAM. If there is no match, failure is signalled, which causes *failure-backtracking* to the most recent *choice-point* (discussed further below) or to the top-level if there is none. Successful completion of an equality assertion causes normal *return* to its caller, and is accompanied by deletion of the corresponding environment record.

If there are no (applicable) equality assertions for a given call, control transfers to any applicable subset assertions. If there are no applicable subset assertions either, failure-backtracking is initiated. The multiple subset assertions that match a given call and the multiple a-c matches within a single subset assertion are attempted sequentially—depth-first computation of subsets is a complete strategy because $\cup$ is strict. We create a *choice-point* record on the control stack to keep track of these alternatives. A single choice-point can record multiple *branch-points* during a-c matching; for example, {{`h1` | `t1`} | {`h2` | `t2`}} has three branch-points, one for each occurrence of "|". The number of branch-points is known at compile-time.

When invoking a function defined by subset assertions, we distinguish two modes of calls: `call-one` and `call-all`. The former is used to call a function—such as `perms`—that appears as an argument to a function—such as `distr`—that distributes over union in this argument; otherwise the latter is used. An *environment* record is created if the subset assertion matching this call has at least one call in its body. In other words, all variables within a subset assertion are assumed to be permanent if the assertion has any function call.

If a subset assertion is invoked by a `call-all` instruction, each successful completion of the assertion causes *success-backtracking* to the most recent choice-point; if it is invoked with a `call-one` instruction, each successful completion causes an *exit* back to the caller. The compiled code for each subset assertion ends with a `collect?` instruction, which tests a 'mode' register to determine whether to initiate success-backtracking or exit—the environment record is not deleted at this time. Once all branch-points within a choice-point have been exhausted, the next subset assertion that matches the call is entered, and the current environment record is deleted. As each subset is computed, it is added to the overall set after removing duplicates. When *failure-backtracking* transfers control to a *choice-point*, the subset computed for this path is assumed to be empty, and execution continues as if success-backtracking had occurred.

Note that the heap is not retracted upon success-backtracking, because the data-structures created along all success backtrack paths are collectively needed. The heap is retracted upon failure backtracking. Garbage collection—not discussed in this paper—is needed to reclaim inaccessible objects in the heap.

3.2 Instruction Set

The state of a SEL program is given by the content of the data areas, as well as certain registers. The following registers and their intended use are identical to that of the WAM: `P`, current program code pointer; `CP`, continuation program code pointer; `E`, last environment pointer; `B`, last choice-point; `A`, top of stack pointer; `H`, top of heap pointer;

`HB`, heap backtrack pointer; `S`, structure pointer (to top of heap); `A1, A2, ...`, argument registers; and `X1, X2, ...`, temporary variables.

In addition, we need the following new registers: `M`, mode of the current call; `CB`, current branch-point; and `B1, B2, ...` branch-point registers.

Similar to the WAM, there are several classes of instructions: *get*, *put*, *store*, *match*, *procedural*, and *indexing*. The main differences are the following:

(i) WAM's *unify* instructions have been replaced by *match* and *store* instructions. The 'read' and 'write' modes of WAM's *get* instructions for lists and structures can be identified at compile-time. All uses of WAM's *get* and *unify* in the 'read' mode are replaced by *get* and *match* instructions; all uses of WAM's *get* and *unify* in 'write' mode are replaced by *store* instructions. All uses of WAM's *put* and *unify* instructions are replaced by *put* and *store* instructions.

(ii) For sets, in addition to the usual *get*, *put*, and *store* instructions, we have the following new instructions: `adj_set_head`, `adj_set`, and `adj_set_with_copy`. These are used immediately after a `get_set` instruction. The difference between `adj_set_head` and `adj_set` is that the former does not construct the remainder of the set. In the latter case, the n different remainders of an n-element set are constructed in O(n) space, using destructive modification. Each invocation of the `adj_set` instruction constructs only one of the remainders. The `adj_set_with_copy` instruction is used when the remainders cannot be constructed in O(n) space. All three instructions establish branch-points, by setting the `CB` and branch-point registers appropriately.

(iii) The procedural instructions of the WAM are augmented with the `call-one`, `call-all`, and `collect?` instructions described earlier. The `collect?` instruction is responsible for constructing the resulting set and removing duplicates, in case the mode register indicates a `call-all` invocation.

(iv) The indexing instructions differ from the WAM in that they do not create choice-points. Choice points are created explictly with a `save_choice_point` instruction. We use `try_equ_else` instructions to link equality assertions, and `try_sub_and` to link subset assertions.

We use a `switch_on_ground_term` instruction for indexing equality and subset assertions, with four cases: constant, list, structure, and set.

We conclude the description of the implementation by showing how the two assertions for `perms` are compiled with these instructions. Each line of the compiled code is commented at the end by showing the corresponding program fragment that it implements. Note that the address of the result of a function is passed as an extra argument (the last), and that the set {[ ]} is represented as {[ ] | { }}.

```
perms/2:  switch_on_ground_term C1, fail, fail, C2

C1:       get_empty_set A1             % perms({ }) =
          store_set A2                 %   {
          store_constant [ ]           %   [ ] |
          store_constant { }           %   { }}
          proceed

C2:       allocate
          get_set A1, Y0               % perms({
          adj_set Y0
          match_variable Y1            %        x |
          match_variable Y2            %        t}) ⊇
          get_variable Y3, A2          %   v1
          save_choice_point            % :-
          put_value Y2, A1             % perms(t) ⊇
          put_variable Y4, A2          %   v2,
          call-one perms/2
          put_value Y1, A1             % distr(x,
          put_value Y4, A2             %       v2) =
          put_value Y5, A3             %   v3
          call-all distr/3
          collect?  Y3, Y5             % v1 := v1 ∪ v3
```

4. Conclusions

The two main ideas behind subset-logic programming are: (i) programming with subset and equality assertions, and (ii) computing with restricted a-c matching and innermost reduction. We have illustrated the paradigm through examples, and shown that it is practical by sketching how it can be efficiently implemented using existing technology. The term ‘subset-logic programming’ was coined by O’Donnell in his recent paper [9]. Our approach, which was independently developed, differs from that of [9] in two important respects: the emphasis on (restricted) a-c matching, and the provision of functions that are not always required to distribute over union over their argument sets.

There is an acknowledged need for a declarative approach to sets in both functional and logic programming [12, 8]. Recent approaches, surveyed in [6], differ from ours in that they are motivated by the need to provide sets in relational databases, and are based on Horn clauses rather than subset assertions. Compilation concerns are emphasized in [11], where an efficient scheme for commutative-idempotent matching is proposed. While our approach to subset-logic programming was motivated by the need to collect all solutions, further work is needed to integrate subset-logic and predicate-logic programming, so that Prolog’s ‘setof’ construct can be captured declaratively. We are also trying to automatically characterize as much as possible (at compile-time) the confluence of equality assertions with a-c constructors and also the distribution of functions over union. At the time of finalizing this paper, we have nearly completed the implementation of SEL using the approach described here [7].

Acknowledgments

Thanks to David Plaisted for discussions which helped formulate these ideas, and to the anonymous referees for their detailed comments and suggestions. This research is supported by grant DCR-8603609 from the National Science Foundation and contract N 00014-86-K-0680 from the Office of Naval Research.

References

[1] D. Benanav, D. Kapur, and P. Narendran, “On the complexity of matching problems,” In *Rewriting Techniques and Applications*, pp. 417-429, Dijon, May 1985.

[2] K. L. Clark, “Negation as Failure,” In *Logic and Data Bases*, Ed. H. Gallaire and J. Minker, Plenum Press, New York, 1978, pp. 293–322.

[3] J. Jaffar, J.-L. Lassez, “Constraint Logic Programming,” In *14th ACM POPL*, pp. 111-119, Munich, 1987.

[4] B. Jayaraman and D.A. Plaisted, “Functional Programming with Sets,” In *Third Int'l Conference on Functional Programming Languages and Computer Architecture*, pp. 194-210, Portland, 1987.

[5] R.A. Kowalski, “Predicate Logic as Programming Language,” IFIP Proc., 1974, pp. 569-574.

[6] G.M. Kuper, “On the Expressive Power of Logic Programming Languages with Sets,” In *7th ACM PODS*, pp. 10-14, 1988.

[7] A. Nair, “Compilation of Subset-logic Programs,” M.S. Thesis, University of N. Carolina at Chapel Hill, expected August 1988.

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

[9] M. J. O'Donnell, “Term-rewriting Implementation of Equational Logic Programming,” In *Rewriting Techniques and Applications*, pp. 1-12, Bordeaux, 1987.

[10] G. Plotkin “Building-in equational theories,” In *Machine Intelligence* **7**, pp. 73-90, 1972.

[11] O. Shmueli, S. Tsur, C. Zaniolo, “Compilation of Rules Containing Set Terms in a Logic Data Language (LDL),” MCC TR DB-222-87, July 1987.

[12] D. A. Turner, “Miranda: A non-strict functional language with polymorphic types,” in *Conf. on Functional Prog. Langs. and Comp. Arch.*, Nancy, Sep. 1985, pp. 1-16.

[13] D. H. D. Warren, “An Abstract Prolog Instruction Set,” Tech. Note 309, SRI International, Menlo Park, 1983.