Subset-Logic Programming: Application and Implementation

TR88-011

February 1988

Bharat Jayaraman and Anil Nair

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

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.

Tel: (919) 962-1764
E-mail: bj@cs.unc.edu

Abstract

Subset-logic programming is a paradigm of programming with subset and equality assertions. We propose this paradigm as a logical basis for programming with sets. We present a language called SEL to illustrate the approach. The terms of SEL are the usual first-order terms of Prolog, augmented with one associative-commutative (a-c) constructor, ∪, for defining sets. Computationally, we treat assertions as one-way rewrite rules, where the matching used is a restricted form of associative-commutative matching. Unlike Prolog's unification, a-c matching could produce multiple matching substitutions, which can effectively serve to iterate over the elements of sets, thus permitting many useful set operations to be stated non-recursively. We also describe the implementation of SEL. We show how WAM-like instructions can be used to compile SEL programs. Because matching rather than unification is used, the 'read' and 'write' modes of 'get' instructions can be identified at compile-time. Two forms of backtracking occur: in addition to backtracking upon failure, the implementation also backtracks upon success in order to collect all elements of a set. An important property of a SEL function is whether or not it 'distributes over nondeterminism' 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.

† 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.

1. Introduction

The term 'logic programming' is often taken to be synonymous with predicate-logic programming, owing to the latter's simple semantics [K74] and the success of Prolog [WPP77]. In recent years, other forms of logic programming have been proposed, most notably equational-logic [O85] and constraint-logic programming [JL87]. We contribute another such approach in this paper, called subset-logic programming. The main motivation for our work was to provide a rigorous basis for programming with sets. Existing approaches, such as the 'setof' construct of most Prolog systems [N85] or the relative-set construct of functional languages [T85], are not supported by an underlying logic, although they are very useful in practice.

In our approach, a program is a collection of two kinds of assertions:

(i) equality assertion: *f(terms)* = *expression*

(ii) subset assertion: *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 the 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 term t such that t = *expr* is a logical consequence of the program assertions.

The language vehicle we present 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 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 $\cup$. 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 actually associative-commutative (a-c) matching [P72], because of the presence of the $\cup$ constructor. Unlike unification, a-c matching could have multiple matching substitutions.

In this paper, we restrict the use of $\cup$ on the left-sides of 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 a-c matching; (ii) formulating problems in terms of sets rather than lists provides more parallelism, because sets relax the sequencing constraint of lists; (iii) nondeterministic search can be specified without the use of 'cuts'; (iv) efficient (non-backtrackable) execution is possible with equations; and (v) checks for duplicate elements in argument sets and formation of intermediate sets can be avoided when operations using these sets 'distribute over nondeterminism' (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 [W83] are very appropriate for the compilation of 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. Because the underlying implementation model for SEL is so similar to the WAM model, we believe that combining predicate-logic and subset-logic programming would be practically feasible.

We described the language SEL, and its declarative and operational semantics in an earlier paper [JP87]. The main objective of this paper is to show the relevance of SEL for 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 and possible extensions.

2. Subset-logic Programming

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

term ::= *atom* | *variable* | { } | {*term*} | *term* ∪ *term* | *constructor*(*terms*)
terms ::= *term* | *term* , *terms*
expr ::= *term* | {*expr*} | *expr* ∪ *expr* | *constructor*(*exprs*) | *function*(*terms*) |
if *expr* then *expr* else *expr*
exprs ::= *expr* | *expr* , *exprs*

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 {...} notation for sets, e.g. {1, 2, 3}, and also use {h | t} to refer a non-empty set, one of whose elements is h and the remainder of the set is t. Thus, {h | t} ≡ {h} ∪ t. The set {1, 2, 3} may be represented as {1} ∪ {2} ∪ {3} ∪ { }. Other permutations, such as {2} ∪ {1} ∪ {3} ∪ { }, {1} ∪ {3} ∪ {2} ∪ { }, etc., represent the same set. Lists and sets may be freely combined in SEL. The constructor ∪, which stands for set union, is associative and commutative, with the properties, x ∪ x = x (idempotence) and x ∪ { } = x (identity), where { } stands for the empty 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 $t_1\theta =_{ac} t_2$? Note that the equality $=_{ac}$ is based only the associative and commutative properties, but not the idempotent property. Thus, for example, matching {h | t} with {1, 2, 3} cannot yield the matching substitution {h ← 1, t ← {1, 2, 3}}. However, {1} can match {h | t}, yielding {h ← 1, t ← { }}.

Plotkin [P72] was perhaps the first to study a-c matching, which he used for building-in equality theories in resolution theorem-provers. To the best of our knowledge, a-c matching has not been previously considered for practical logic programming. For both programming and implementation simplicity, we propose to disallow *explicit* use of the ∪ constructor on the *left-sides* of SEL assertions. Instead, we permit arbitrary combinations of patterns of the form

{*term* | *term*}.

While some expressive power is sacrificed by this restriction, most practical cases are unaffected. This restriction turns out to be very important for compilability of SEL programs.

Below we present a Prolog program to specify more precisely the behavior of the matching algorithm, assuming the above restriction. 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* or *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 [JP87] in terms of rewrite rules.

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

`distr(x,{h | t})` $\supseteq$ `{[x | h]}`

all three matches are considered, namely, {x ← 10, h ← 1, t ← {2,3}}, {x ← 10, h ← 2, t ← {1,3}}, and {x ← 10, h ← 3, t ← {1,2}}. The right-side of the assertion for `distr`, namely {[x | h]}, is then fully reduced for each of these matches, and the union of the fully reduced results is defined as the value for `f({1,2,3})`. Thus, the value returned in this

case would be {[10|1], [10|2], [10|3]}. 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, i.e., $x \cup \bot = \bot$. However, because of the closed-world assumption, if any one these reductions terminates with a non-term expression ($\top$), its result can be assumed to be { } for the purpose of the union, i.e., $x \cup \top = x$.

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 [JP87], 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 $x \neq$ `true` $\wedge\ x \neq \bot$.

That is, the conditional expression implements a form of *negation by failure* [C78].

2.3 Examples of SEL Programs

Append:

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

First-order functional programming can be carried out in the usual way with equations, as the above example suggests.

Set Intersection:

```
intersect({ },s) = { }
intersect(s,{ }) = { }
intersect({h | _ },{h | _ }) ⊇ {h}
```

Finding common elements in the two sets is finessed by a-c matching. 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.

Relative Set Abstraction:

```
all-fp({ }) = { }
all-fp({x | _ }) ⊇ if p(x) then {f(x)} else { }
```

The above assertions serve to effectively define the relative set construct, $\{f(x) \mid x \in S \wedge p(x)\}$. Here, a-c matching is used to iterate over the elements of the argument set.

Permutations:

```
perms({ }) = {[ ]}
perms({x | t}) ⊇ distr(x,perms(t))
distr(x,{ }) = { }
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 $\{1,2,3,4\}$, 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

The above examples serve as a basis for the following more general points:

1. Note that `intersect`, `all-fp`, and `distr` are stated non-recursively. We similarly defined several other useful operations in our recent paper [JP87]. This is one of the strengths of a-c matching. All of the programs using relative set-abstraction in Miranda [T85] may be macro-expanded into SEL assertions, in the manner shown in the `all-fp` example.

2. Formulating problems, e.g. permutations, with sets rather than lists allows more parallelism. Note that comparable declarative formulations of this problem in existing functional and logic languages cannot achieve as much parallelism as the SEL formulation because they sequentialize the generation of permutations by treating the answer as a *list* of permutations. Because of one-way matching, the or-parallelism arising from a-c matching does not incur the problem of having to maintain multiple bindings of unbound variables in the parent environment [HCH87, SW87].

3. We say that an operation `f` *distributes over nondeterminism in the i-th argument* iff

$$f(\ldots, x \cup y, \ldots) = f(\ldots, x, \ldots) \cup f(\ldots, y, \ldots)$$

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 nondeterminism. Functions, such as `distr` and `intersect`, that are defined in terms of the elements of the set, do distribute over nondeterminism. There are two benefits of knowing that a function distributes over nondeterminism 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 nondeterminism.

3. Implementation

We 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 [W83].

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.

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

Note that the operation `distr` distributes over nondeterminism (in its second argument), but `perms` does not. This is distinguished in the compiled code by the use of $\supseteq$ in flattening `perms`, and the use of $=$ in flattening `distr`. 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 possible for equality assertions (because they are deterministic) but not for subset assertions. 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 ∪ 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 nondeterminism 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, we use four new instructions: `get_empty_set`, `get_set`, `get_set_head`, and `put_set` instructions. The difference between `get_set` and `get_set_head` is that the latter does not construct the remainder of the set. In the former case, the n different remainders of an n-element set are constructed in a total of n extra words, rather than $O(n^2)$ extra words. Each invocation of the `get_set` instruction constructs only one of the remainders. Both these 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 `set_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                  % perms({
match_variable Y1           %        x |
match_variable Y2           %        t}) ⊇
get_variable Y3, A2         %   v1
set_choice_point            % :-
put_value Y2, A1            % perms(t) ⊇
put_value Y4, A2            %   v2,
call-one perms/2
put_value Y1, A1            % distr(x,
put_value Y4, A2            %       v2) =
put_value Y5, A3            %   v1
call-all distr/3
collect?  Y3, Y5            % Y3 := Y3 ∪ Y5
```

4. Conclusions

There is an acknowledged need for a declarative approach to sets in both functional and logic programming [T85, N85]. Our work represents an attempt to fulfil this need. The two main ideas behind subset-logic programming are: (i) programming with subset and equality assertions, and (ii) computing with a-c matching and rewriting. We presented the formal semantics of subset-logic programming in an earlier paper [JP87]. In this paper, we have illustrated the paradigm through examples, and shown that it is practical by sketching how it can be efficiently implemented using existing technology. We are in the process of implementing the language SEL described in this paper.

We have tried to be conservative in our design, in that we have tried to provide the smallest set of features that will be declarative, useful, and efficiently implementable. Many extensions appear to be possible: non-strict constructors, absolute set-abstraction, higher-order features, and also the integration of subset-logic and predicate-logic programming. We are at present investigating these extensions. 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 nondeterminism.

References

[C78] 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.

[HCH87] B. Hausman, A. Ciepielewski, S. Haridi, "OR-Parallel Prolog Made Efficient on Shared Memory Multiprocessors," In *1987 Symp. on Logic Prog.*, pp. 69-79, San Francisco, 1987.

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

[JP87] B. Jayaraman and D.A. Plaisted, "Functional Programming with Sets," In *Third Int'l Conference on Functional Programming Languages and Computer Architecture*, pp. 194-210, Portland, 1987.

[K74] R.A. Kowalski, "Predicate Logic as Programming Language," IFIP Proc., 1974, pp. 569-574.

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

[O85] M. J. O'Donnell, "Equational logic as a programming language," M.I.T. Press, 1985.

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

[SS86] L. Sterling and E. Shapiro, "The Art of Prolog," MIT Press, 1986.

[SW87] K. Shen and D.H.D. Warren, "A Simulation Study of the Argonne Model for OR-Parallel Execution of Prolog," In *1987 Symp. on Logic Prog.*, pp. 54-68, San Francisco, 1987.

[T85] D. A. Turner, "Miranda: A non-strict functional language with polymorphic types," in *Conf. on Functional Prog. Langs. and Comp. Arch.*, Nancy, France, Sep. 1985, pp. 1-16.

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

[W83] D. H. D. Warren, "An Abstract Prolog Instruction Set," Tech. Note 309, SRI International, Menlo Park, October 1983.