# SUBSET LOGIC PROGRAMS AND THEIR IMPLEMENTATION

BHARAT JAYARAMAN AND KYONGHEE MOON

▷ This paper discusses the design and implementation of a set-oriented logic programming paradigm, called subset-logic programming. Subset-logic programs are built up of three kinds of program clauses: subset, equational, and relational clauses. Using these clauses, we can program solutions to a broad range of problems of interest in logic programming and deductive databases. In previous research, we developed the implementation of subset and equational program clauses. This paper substantially extends that work, and focuses on the more expressive paradigm of subset and relational clauses. This paradigm supports *setof* operations, transitive closures, monotonic aggregation as well as incremental and lazy enumeration of sets. Although the subset-logic paradigm differs substantially from that of Prolog, we show that few additional changes are needed to the Warren Abstract Machine (WAM) to implement the paradigm and that these changes blend well with the overall machinery of the WAM. A central feature in the implementation of subset-logic programs is that of a *monotonic memo-table*, i.e., a memo-table whose entries can monotonically grow or shrink in an appropriate partial order. We present in stages the paradigm of subset-logic progams, showing the effect of each feature on the implementation. The implementation has been completed, and we present performance figures to show the efficiency and costs of memoization. Our conclusion is that the monotonic memo-tables are a practical tool for implementing a set-oriented logic programming language. We also compare this system with other closely related systems, especially XSB and CORAL.

**Keywords:** subset and relational program clauses, sets, set matching and unification, memo tables, monotonic aggregation, Warren Abstract Machine, run-time structures, performance analysis ◁

*Address correspondence to* Bharat Jayaraman, Department of Computer Science, State University of New York at Buffalo, 226 Bell Hall, Buffalo, NY 14260-2000, E-Mail: bharat@cs.buffalo.edu




655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

## 1. Introduction

This paper describes the implementation of a logic programming paradigm based upon three kinds of program clauses: equational, subset, and general relational clauses. This paradigm is called *subset logic programming*, and it has been the subject of our investigations over the past several years [9, 10, 11, 12, 13, 14, 15, 18, 23, 25]. The particular language that was implemented is called *SuRE*, which stands for *Su*bsets, *R*elations, and *E*quations[1]. While equational and relational clauses are well-known in functional and logic programming respectively, subset clauses are relatively a recent development. Hence we first motivate our interest in subset clauses, describe the new implementation issues they raise, and outline our proposed solutions.

### *1.1. Subset Logic Programming*

Our interest in subset-logic programming stems from the fact that it provides a declarative and efficient means of working with sets, a feature that has received considerable recent interest in logic programming and deductive databases [2, 4, 5, 15, 16]. While sets are ubiquitous in applications of logical reasoning, practical functional and logic languages (such as ML or Prolog) do not support *bona fide* sets, apparently due to the difficulty of implementing them efficiently. For example, Prolog's `setof` actually constructs an ordered list, not a set. Our proposed constructs subsume the uses of `setof` and provide several additional capabilities. The central feature of our work is the *subset clause*, which has one of two forms:

*f(terms)* `contains` *expr*
*f(terms)* `contains` *expr* `:-` *condition*

where *terms* are built up from constants, variables, and data constructors; *expr* may in addition contain user-defined functions; and *condition* is a sequence of one or more equational, relational, or membership goals. The syntactic details of *terms*, *expr*, and *condition* are given in section 2.

Informally, the declarative meaning of an unconditional subset clause is that, for all its ground instantiations, the function $f$ applied to its argument terms is a superset of the ground set denoted by the expression on the right-hand side. For a conditional subset clause, in addition the appropriate instance of *condition* must be true. In general, multiple subset clauses may be used in defining some function $f$. By providing subset clauses with a *collect-all* capability, the meaning of a set-valued function $f$ applied to ground terms is equal to the union of the respective sets defined by the different subset clauses for $f$.

An interesting aspect of subset-logic programs is that often a function may be circularly defined using subset clauses. In these cases, it is natural to define the meaning of the function call as the smallest set that satisfies the circular constraints. In general, a program has a well-defined meaning if it obeys a *local stratification* condition in which all circularly defined function calls must depend upon one another through *subset-monotonic* functions, i.e., monotonic with respect to the subset ordering. We illustrate such programs in section 2. A more formal account of the theory of subset-logic programming is given in [9, 23, 24].

[1] To wit, SuRE is the affirmative answer to the question: Can programming be declarative and practical?

Subset clauses can be used in combination with both equational and general relational clauses. The combination of equational and unconditional subset clauses is called *subset-equational* programming. This is a purely functional paradigm, and its implementation issues were mostly discussed in reference [11]. However, these papers do not discuss circular subset constraints and their attendant implementation problems. This paper concentrates on the combination of relational and subset clauses (both unconditional and conditional), a paradigm which may be called *subset-relational* programming. Relational clauses are of the form:

*p(terms)*
*p(terms)* `:-` *condition*

where *terms* and *condition* are as described earlier. The paradigm of subset-logic programming is essentially a union of the subset-equational and subset-relational paradigms. In order to keep the paper self-contained, we include a brief coverage of relevant parts from our earlier work [11].

In the subset-logic paradigm, sets are constructed using two novel set constructors, {X\T} and {X/T}. The constructor {X\T} matches a set S such that X ∈ S and T = S - {X}, i.e., the set S with X removed [11]. This constructor can be used only on the *left-hand sides* of subset clauses and it helps decompose a set into *strictly smaller* subsets—this is why we do not use the more familiar ∪ constructor for pattern-matching with sets. Set patterns made up from this constructor help finesse iteration over sets and thereby minimize the number of procedure calls. The constructor {X/T}, on the other hand, is equivalent to {X} ∪ T, and it is generally used on the *right-hand sides* of subset clauses for including an element X in a set T—using {X\T} on the right-hand sides of subset clauses would result in failure if X ∈ T. The {X/T} constructor has also been advocated by several other researchers in logic programming and deductive databases [4, 5]. (In [11] we used a single constructor {_|_}, which meant {X\T} on the left-hand sides of clauses, and meant {X/T} on the right-hand sides of clauses.) We illustrate the use of both constructors in section 2. All functions defined by subset clauses are invoked with ground arguments (possibly ground sets), and hence we must use *set-matching* in these cases. On the other hand, the use of a set constructor in the head of a relational clause necessitates *set-unification*.

Subset clauses have many uses in the logic programming context: They serve as a declarative alternative to many uses of Prolog-like *mode* declarations as well as those uses of Prolog's `assert` and `retract` that correspond to implementations of memo-tables or collection of results from alternative search paths, as in the `setof` construct. By re-formulating a relation as a set-valued function, one not only specifies mode information declaratively, but also gains the flexibility of operating on the resulting set incrementally or collectively. In the latter case, one further has the flexibility of working lazily or eagerly. We showed in our earlier work [11, 14] that when a function *distributes over union* in one of its arguments, we can operate element-at-a-time with respect to this argument, and thereby avoid forming an intermediate set as well as avoid the check for duplicates in this argument. This semantic property of a function can be specified by the programmer as an annotation, but the property can also be inferred automatically by the SuRE compiler in most practical cases [20].

Subset clauses are also useful in the deductive database context: They help render clear and concise formulations to problems involving aggregate operations and

recursion in database querying. An aggregate operation is a function that maps a set to some value, e.g., the maximum or minimum in the set, the cardinality of this set, the summation of all its members, etc. This has been a topic of considerable interest in the deductive database literature recently [8, 17, 21, 31, 32, 34]. In considering the problems with various semantic approaches, Van Gelder notes that, for many applications in which the use of aggregates has been proposed, the concept of *subset* is what is really necessary [34]. Moreover, for many problems requiring aggregate operations to be performed, the concept of monotonic functions is necessary [31]. We have recently shown that the subset-logic paradigm and its generalization (i.e., replacing subset clauses by partial-order clauses) are particularly well-suited for programming such operations because the concepts of aggregation, subset, and monotonicity are more naturally expressed in terms of functions rather than predicates [25].

### *1.2. Implementation Issues*

The Warren Abstract Machine (WAM) [1, 35], designed by D.H.D. Warren, has proven to be a robust framework for implementing Prolog as well as several variants of the language [6, 11, 22, 33]. Our basic strategy is to extend the WAM with new instructions and run-time structures in order to implement the new control regime as well as the new matching and unification operations of subset-logic programs. It turns out that, even though the subset-logic paradigm differs substantially from Prolog, few additional instructions, registers, and storage structures are needed and these blend in well with the overall design of the WAM. Our decision to use the WAM model implies that we are using a top-down (as opposed to a bottom-up) execution model for subset-logic programs. Since memoization plays a crucial role in our implementation, the use of a top-down execution model has no inherent disadvantages. On the contrary, recent work indicates that a WAM model with memoization shows an order of magnitude performance improvement over bottom-up methods [33] for the class of definite clause (relational) programs. This improvement is due to the efficiency in data-structures and backtracking of the WAM.

Due to the presence of sets in the language, the efficient realization of set-matching and set-unification is a central implementation problem. The implementation of both these operations has been treated in earlier papers—set-matching is described in [11] and set-unification in [6]—and therefore we discuss them only briefly in this paper. Both operations can in general result in multiple maximally general matches/unifiers, and hence may cause additional branching in the search process. Set unification also results in new variables being introduced (during unification), an issue that doesn't arise in standard, first-order unification. Our experience shows that for many practical subset-logic programs making use of sets in relational clauses, set-unification reduces to set-matching, and these cases can be detected through static analysis [20]. Thus, the knowledge of the context in which set terms are used helps obtain a more efficient implementation. One implementation of set-unification has been described in [6], but we take a different approach in this paper. The main difference from [6] is that the implementation of set-unification is done through constraint management, whereas our language does not support constraints, and hence we implement set unification within the basic WAM structure, but with an extended form of a push-down list. In previ-

ous work, [11] showed that set-matching can be compiled in terms of an extended WAM instruction set, and that remainder sets in patterns of the form {X\T} can be efficiently constructed and represented. To keep track of the branching within set-matching and set-unification, we introduce a *branch point* record on the control stack of the WAM.

The control cycle of the abstract machine for subset-logic programs differs substantially from that of the WAM. Still we are able to accomodate it with a few simple additions. In contrast with Prolog programs, branching in subset-logic programs occurs because (the left-hand sides of) multiple subset clauses may match a given call and also because any one clause may match in multiple ways due to the presence of set constructors. Thus the body of a subset clause would in general have to be re-executed several times, each time with a different match, and the union of the resulting sets for these different executions would have to be formed. There are four different *modes* in which a set-valued function (defined by a subset clause) may be called: *eager* evaluation, *incremental* evaluation, *memoized* evaluation, and *lazy* evaluation. When a function is invoked eagerly (using an equational goal) the entire set for the function call is formed. This is accomplished by traversing the search tree exhaustively in depth-first order with backtracking. When a function $f$ is invoked via a membership goal or if $f$ occupies an argument position of another function that distributes over union in this argument, then the search tree for the function call is not traversed exhaustively, but rather incrementally, to yield one element at a time to the caller of $f$. Incremental evaluation does represent the entire set explicitly, and therefore contrasts with lazy evaluation which may form the entire set but does so lazily. A fundamental difference between lazy evaluation in the subset-logic paradigm and that in a conventional functional paradigm owes to the fact that the unevaluated portion of a lazy set might be some control point in the search tree of a relational program. For the sake of brevity, we do not discuss in detail the implementation of lazy sets in this paper; the interested reader may consult for the details [20].

Memoization in subset-logic programs differs from the way it is used in both functional and logic programming languages: In functional programming, memoization is used to detect dynamic common subexpressions [7], and serves to obtain an efficient implementation of dynamic programming algorithms. In addition to this motivation, memoization in logic programs is used to detect *circular* loops that arise in programs for problems such as transitive closure [33, 36]. In subset-logic programs, memoization is used for detecting dynamic common subexpressions as well as for loop detection. The latter capability is needed when circular function calls arise because of circular subset constraints. However, simple loop detection is insufficient to realize the semantics of subset-logic programs: When circular function calls depend upon one another through subset-monotonic functions, these calls have to be progressively iterated (or re-executed) until their least/greatest fixed point is reached. This in turn requires memo-table entries to be monotonically updated—as we shall explain in more detail later. In order to record where a memo-table entry was consulted, we store a *lookup point* record on the WAM control stack. By chaining together all such lookup points, we can direct re-execution to the most recent lookup point. The above scheme is sufficient for unconditional subset clauses. For a conditional subset clause—which might contain relational goals in their bodies—in order to restore the environment to the correct state for re-execution, we need to protect the bindings of variables that might have be reset during backtracking

amongst relational goals. Hence we introduce a new structure called a *redo trail*, in which we save the bindings of such variables at the time a lookup point is created. These bindings are recovered when a re-do is to be performed.

We represent the memo-table by a hash-table in order to achieve fast look-up. Note that memo-table entries for subset-logic programs will always contain ground terms (possibly ground sets), because the arguments and results of functions will be ground. That is, we memoize only function calls, and not predicate calls. In this respect, our approach differs from that of [33]. However, as noted earlier, for many problems of involving aggregation in deductive databases, such as shortest path, company controls, etc., it suffices to work with functions and ground terms.

### *1.3. Experimental Results and Comparisons*

We provide performance figures to illustrate the key new features of our implementation. While this implementation should be regarded as a prototype, we nevertheless present performance figures in order to demonstrate certain properties. For example, we show that the overhead of memoization in certain kinds of subset-logic programs is minimal. This result is similar to the one reported in [33] for Prolog programs, namely, that tabled execution for programs that have no redundant computation is comparable with ordinary WAM execution. We show that the use of monotonic memo-tables to implement dynamic programming algorithms can be a more efficient way than using pure memo-tables. We report performance figures for problems requiring monotonic aggregation in deductive databases—our extensional database is an in-memory database, as is the case with Prolog databases.

We compare the performance of SuRE with two closely related systems, XSB and CORAL, on typical problems in deductive databases. These systems share some features with each other, but there are differences in their respective language and computational models. Both XSB and CORAL can evaluate circular calls without incurring nontermination problems as in Prolog.

1. The XSB system, developed by D.S. Warren and researchers at Stony Brook, implements a language that is very similar to Prolog [27, 37]. The operational semantics of XSB is based upon SLG resolution, a table-oriented resolution method that combines SLD resolution with memoization [33]. The X in XSB stands for extension table, or memo-table, while the G in SLG stands for general clauses, as opposed to definite clauses. XSB applies SLD resolution for non-tabled predicates, and memoization for predicates that are declared by the programmer as tabled predicates. Although both SuRE and XSB use memo-tables to detect circular calls, XSB memoizes calls to predicates whereas SuRE memoizes calls to functions. As a result, in XSB a single memo-table entry can have several different answer clauses, and each answer clause has to be resolved against the subgoals. A fixed point is reached when there are no subgoals to be resolved upon. The answer clause resolution of XSB is different from the table look-up operation of SuRE. Memoization in SuRE is restricted only to functions, and a given function call has only one answer. However, this answer can be a set, which, in general, is determined by a process of monotonically updating memo-table entries until a fixed point is reached. In XSB, a new memo-table entry is created whenever a call to a tabled subgoal is made that is not a variant of a previous tabled

subgoal. When a selected subgoal is already in the table and there is no answer, the operation of that subgoal is suspended (this is similar to creating a look-up point record in SuRE) and backtracking to the previous goal occurs (in contrast, SuRE would assume the default value and proceed). A suspended goal resumes an operation when an answer to it becomes available and resolved via an answer clause resolution. Like Prolog, XSB does not support sets, and it makes use of the second-order predicates `setof` or `bagof` for collecting all solutions into a list.

2. CORAL is a deductive database system developed by R. Ramakrishnan and researchers at Wisconsin [28]. CORAL stands for COntrol, Relations, And Logic. Unlike SuRE and XSB, CORAL uses bottom-up evaluation with *magic rewriting*. The declarative language that it implements is essentially definite clauses with negation and multiset-generation capabilities. CORAL supports multisets, or bags, and a limited form of *grouping*, which is similar to set-collection in SuRE. Although grouping allows construction of a multiset, this multiset may be compared only against another ground multiset term or assigned to a variable. General matching and unification of sets are not supported [28]. This may be contrasted with SuRE which supports nonground set-terms. Although the main strategy is bottom-up evaluation—'materialization' in database terms—CORAL also supports Prolog-like execution through a pipelining annotation, a predicate-level annotation, and a rule-level annotation. Even without user-specified annotations, the CORAL compiler attempts to perform some optimizations. An important feature of CORAL is its module system, which is key to incorporating compile-time as well as run-time optimizations. CORAL sometimes shows marked differences in performance between the case when a single module is used for both program and data and the case when separate modules are used for program and data: The idea is that, by moving many rules out of a module, fewer rules are involved during bottom-up evaluation [28].

The rest of this paper is organized as follows: section 2 presents the syntax and examples of subset-logic programs; section 3 develops the abstract machine for subset-logic programs in stages, starting with a basic abstract machine for subset clauses and then showing the changes needed to accomodate each additional feature on the paradigm; section 4 presents the new instructions of the abstract machine, and examples of compiled code; section 5 presents performance figures from our current implementation and comparisons with the XSB and CORAL systems; finally, section 6 presents our conclusions and areas of further work. We assume some familiarity with the implementation of logic programs, especially the Warren Abstract Machine [1, 35].

## 2. Subset Logic Programs: An Informal Introduction

A subset-logic program is built up of equational, subset, and general relational clauses. We do not treat equational clauses in this paper, as they have been fully discussed in [11]. Our focus is on subset and relational clauses, each of which may be unconditional or conditional. We begin with unconditional subset clauses in section 2.1, and discuss set matching, distribution over union, memoization, and monotonic

memo-tables in this context. We then describe in section 2.2 conditional subset and relational clauses, and discuss `setof` operations and monotonic aggregation. We also introduce partial-order clauses in that subsection.

## *2.1. Unconditional Subset Clauses*

Unconditional subset clauses are of the form

*f(terms)* `contains` *expr*

where each variable in *expr* must also occur in *terms* in order for an unconditional clause to be well-formed. The syntax of *terms* is:

*terms* ::= *term* | *term* , *terms*
*term* ::= *variable* | *constant* | *constructor*(*terms*) | *set*
*set* ::= `phi` | *variable* | {*term**set*}

Our lexical convention in this paper, as in Prolog, is to write constants and (non-set) constructors starting with lowercase letters, and variables starting with uppercase letters. A *ground term* is a *term* without any variables. Ground terms are built up from constants and constructors and stand for data objects of the language. The syntax of *expr* is:

*expr* ::= *variable* | *constant* | `phi` | {*expr*/*expr*} |
*constructor*(*exprs*) | *function*(*exprs*)
*exprs* ::= *expr* | *expr* , *exprs*

where a *function* is a non-constructor, or user-defined, symbol, i.e., a symbol appearing at the head of the left-hand side of a subset clause.

Informally, the declarative meaning of a subset clause, *f(terms)* `contains` *expr*, is that, for all its ground instantiations (i.e., replacing variables by ground terms), the function *f* applied to its argument terms contains the ground set denoted by the instance of *expr*. The top-level goal is a ground expression $e$, and its meaning is the ground term $t$ such that $e = t$ is a logical consequence of a *completion* of the program [15, 9]. The *completion* incorporates two assumptions underlying the meaning of subset-equational programs: the *collect-all* assumption and the *emptiness-as-failure* assumption. Given a subset logic program and a ground expression $f(terms)$, if it follows from the program that $f(terms)$ `contains` $s_1$ , ..., $f(terms)$ `contains` $s_n$, for some ground sets $s_1$, ..., $s_n$, and no other sets are contained in $f(terms)$ according to the given program, then the collect-all assumption allows us to infer that $f(terms) = \cup_{i=1,n} s_i$, provided that the resulting set is finite and all reduction sequences from $f(terms)$ terminate. The emptiness-as-failure assumption states that a ground goal $f(terms)$ returns `phi` as its result if *terms* does not match the left-hand side of any subset clause for *f*. We illustrate these ideas below with examples.

### *2.1.1. Iteration via Set Matching*

As noted in section 1, the term {`X`\`T`} matches a set `S` such that `X` ∈ `S` and `T`=`S`-{`X`}, i.e., the set `S` with `X` removed. Thus, a set represented using {__} will not contain any duplicate elements. This constructor has the property that

$\{t_1\backslash\{t_2\backslash s\}\} = \{t_2\backslash\{t_1\backslash s\}\}$,

reflecting the fact that the order of members in a set are irrelevant. We permit $\{t_1\}$ as an abbreviation for $\{t_1\backslash\texttt{phi}\}$. Also, $\{t_1, t_2, \ldots, t_n\}$, for $n > 1$, is an abbreviation for $\{t_1\backslash\{t_2\backslash\{\ldots\{t_n\backslash\texttt{phi}\}\ldots\}$. Similarly, $\{t_1, t_2, \ldots, t_n\backslash T\}$, for $n > 1$, is an abbreviation for $\{t_1\backslash\{t_2\backslash\{\ldots\{t_n\backslash T\}\ldots\}$.

To illustrate, matching $\{\texttt{X}\backslash\texttt{T}\}$ with $\{\texttt{a},\texttt{b},\texttt{c}\}$ gives three different outcomes:

$\texttt{X} \leftarrow \texttt{a}, \quad \texttt{T} \leftarrow \{\texttt{b},\texttt{c}\}$
$\texttt{X} \leftarrow \texttt{b}, \quad \texttt{T} \leftarrow \{\texttt{a},\texttt{c}\}$
$\texttt{X} \leftarrow \texttt{c}, \quad \texttt{T} \leftarrow \{\texttt{a},\texttt{b}\}$

There are only finitely many matches of a set-term $\{\texttt{X}\backslash\texttt{T}\}$ against any set constructed from `phi` and $\{\_\backslash\_\}$. While this matching problem is NP-complete, in practice set patterns have very simple forms, and hence set-matching is a practical tool for iterating over the elements of a set. We illustrate this point with a few examples. The union of two sets can be defined as follows:

```
union(X1,X2) contains X1
union(X1,X2) contains X2
```

By the *collect-all* assumption, we infer that $\texttt{union}(\texttt{X1},\texttt{X2}) = \texttt{X1} \cup \texttt{X2}$. Aside: the term $\{\texttt{X/T}\} = \texttt{union}(\{\texttt{X}\},\texttt{T})$, and hence the constructor $\{\_/\_\}$ need not be treated as a primitive when used on the right-hand sides of clauses.

Often definitions can be stated in a compact, non-recursive manner using subset clauses because much of the iteration over sets is moved into the matching process. For example, the product and intersection of two sets can be defined as follows:

```
setproduct({X\_},{Y\_}) contains {pair(X,Y)}
intersect({X\_},{X\_}) contains {X}
```

When set patterns occur on the left-hand sides of subset clauses, all matches against these patterns are used in instantiating the corresponding right-hand side expression, and the union of the resulting sets is taken as the result. For example, the definition of `intersect` works as follows: For a goal $\texttt{intersect}(\{1,2,3\},\{2,3,4\})$, we get $\texttt{intersect}(\{1,2,3\},\{2,3,4\})$ `contains` $\{2\}$ and $\texttt{intersect}(\{1,2,3\},\{2,3,4\})$ `contains`$\{3\}$. By the collect-all assumption, we infer $\texttt{intersect}(\{1,2,3\},\{2,3,4\}) = \{2\} \cup \{3\} = \{2,3\}$. By the emptiness-as-failure assumption, both `setproduct` and `intersect` will return `phi` if they are called with either argument as `phi`.

The use of remainder sets in set-matching is illustrated by the following function definition, whose input is a set of *propositional clauses* (i.e., a set of set of literals) and whose output is the set of all resolvents:

```
resolvents({{X\S1}, {not(X)\S2} \_}) contains {union(S1,S2)}
```

Here, the remainder sets `S1` and `S2` are crucial in constructing the resolvents. Note that $\{\texttt{A},\texttt{B}\backslash\_\}$ is an abbreviation for $\{\texttt{A}\backslash\{\texttt{B}\backslash\_\}\}$.

Recursive subset clauses are also meaningful and natural, as illustrated by the following program to find the set of all list arrangements of a set (we use the Prolog notation [ ] for the empty list and [X|L] for a nonempty list with head `X` and tail `T`):

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

A detailed explanation of this program appears in [11].

The `perms` definition brings up the issue of nested expressions in the body of a subset clause. Basically these expressions are executed in leftmost-innermost order ("call by value"), and the default strategy is to compute the resulting set for an inner call before executing an outer call. However, this strategy is altered when a function distributes over union in some argument. We say that a function $f$ *distributes over union* in its *i-th* argument if

$$f(\ldots, s_1 \cup s_2, \ldots) = f(\ldots, s_1, \ldots) \cup f(\ldots, s_2, \ldots)$$

where the *i-th* argument is the one shown above and all other arguments remain unchanged on both sides of the equality. Another way to state this property is that

$$f(\ldots, \{x_1, \ldots, x_n\}, \ldots) = \bigcup_{j=1}^{n} f(\ldots, \{x_j\}, \ldots).$$

Thus we see that functions that are defined in terms of the elements of their argument sets distribute over union in these arguments. Examples include `union`, `setproduct`, and `intersect`. On the other hand, functions that compute some aggregate property of a set, e.g., `resolvents` and `perms`, do not distribute over union. (Note that `distr` distributes over union in its second argument.) In recognition of this property, we distinguish two *calling modes* for functions: Under *incremental* evaluation, the resulting set is produced one element at a time, and hence we say that the function is called in *call-one* mode. Under *eager* evaluation, the resulting set is produced all at once, and we say it is called in *call-all* mode. In our current implementation, this property is indicated by a `mode` annotation:

```
mode perms(no), distr(no, yes), intersect(yes,yes).
```

Unlike the mode declaration of Prolog, the above is a declarative (as opposed to a procedural) annotation. Because $f(\ldots, \{x_1, \ldots, x_n\}, \ldots) = \bigcup_{j=1}^{n} f(\ldots, \{x_j\}, \ldots)$, when a function distributes over union with respect to one of its arguments, it suffices to incrementally generate the elements of this argument set. The advantages of this approach are: (a) space is saved by not forming the entire argument set, and (b) time is saved by not checking for duplicates in this argument set. Our experiments show that the use of the mode annotation does yield substantial improvements in the programs that we have encountered; and the overhead of extra function calls is much less than the cost of duplicates checking and intermediate set formation. A more detailed discussion of the effect of this annotation on the performance improvement of programs is given in [11].

### *2.1.2. The Need for Memoization*

There are two different uses of memoization in subset clauses: (i) to detect dynamic common subexpressions, and (ii) to detect circular function calls. While the first use of memoization is motivated purely by efficiency considerations, the second use of memoization is motivated by semantic reasons. The first use commonly arises in recursive subset clauses of the form

$f(\ldots, \{\texttt{X}\backslash\texttt{T}\}, \ldots)$ `contains` $\ldots f(\ldots, \texttt{T}, \ldots) \ldots$

Here the recursion is with respect to the remainder set `T`. An example is provided by the `perms` definition shown earlier. Any call on `perms` with an argument set of size $n$, where $n \geq 2$, will result in $n \times (n-1)/2$ identical pairs of calls on `perms` with argument sets of size $n-2$.

An example of the second use of memoization arises in computing the transitive closure of a relation. The following is a definition of the set of reachable nodes of a graph starting from a given set of nodes (we re-formulate this problem in section 2.2 using an `edge` predicate):

```
reach(S) contains S
reach({X\_}) contains reach(edge(X))
edge(1) contains {2}
edge(2) contains {1}
```

The intended meaning of the above program is clear even when the `edge` function defines a cyclic graph, as shown above. That is, we expect the goal `reach({1})` to terminate with the resulting set $\{1, 2\}$. This is the smallest set that satisfies the constraints of the program. Notice that the goal `reach({1})` results in a cyclic call on itself. We therefore make use of a memo-table to detect this loop, and return `phi` as the result of the recursive call. This value serves as the first approximation of the final answer to the outer call. When the computed set, $s$, for the outer call differs from the value assumed for the inner (cyclic) call, the inner call is re-executed with $s$ as the new approximation. This process is iterated until a fixed point is reached. We refer to this process of re-execution as *re-do*, and we refer to the resulting memo-table as a *monotonic memo-table*.

It turns out that the `reach` program is simple enough so that no re-execution is necessary; the correct answer to the outer call is obtained at the end of the first iteration itself. However, when circular function calls depend upon one another through *subset-monotonic* functions, in general more than one iteration is needed to obtain the correct final answer. We say that a function $f$ is subset-monotonic in its *i-th* if

$$s_1 \subseteq s_2 \Rightarrow f(\ldots, s_1, \ldots) \subseteq f(\ldots, s_2, \ldots)$$

where the *i-th* argument is the one shown above and all other arguments remain unchanged on both sides of $\subseteq$ in the consequent of the implication. We illustrate this case with the following program which defines the *reaching definitions* in a program flow graph, a set which is computed by a compiler during its optimization phase [3]:

```
out(B) contains diff(in(B), kill(B))
out(B) contains gen(B)
in(B) contains allout(pred(B))
allout({P \_}) contains out(P)
```

The program flow graph is defined by the function `pred` which gives the set of immediate predecessor nodes of any given node. The functions `kill(B)` and `gen(B)` are predefined set-valued functions specifying the relevant information for each basic block `B` of the flow graph. The set-difference function `diff` (which is defined in section 2.2.1) is monotonic in its first argument, and hence its use in the body of `out` is legal. (However, the check for monotonicity is not made by the compiler.) The above program is a direct rendering of the flow analysis equations, and the computational model of memoization and re-do is ideally suited to solving such problems. The reader may contrast the above solution to the one given in [3], wherein an imperative program is written to compute the desired sets using a bottom-up strategy.

In general we require subset clauses to obey a *local stratification* condition: All circular function calls must be defined in terms of one another through subset-monotonic functions. Non-circular function-call dependencies are not restricted in any way. Note that this definition of local stratification is similar to the one in [26].

## *2.2. Conditional Subset Clauses*

The syntax of conditional subset clauses is as follows:

*f(terms)* `contains` *expr* `:-` *condition*

where each variable in *expr* appears either in *terms* or in *condition*, and *condition* is a sequence of one or more goals as defined below:

*condition* `::=` *goal* | *goal*, *condition*
*goal* `::=` *p(terms)* | `not` *p(terms)* | *f(terms)* `=` *term*

The declarative reading of a conditional clause is similar to that for unconditional clauses, in that it is given in terms of ground instances of a clause, except that the ground instance of the head of the clause is considered true if the corresponding instance of the condition in the body is true. The meaning of a ground expression $e$ is the ground term $t$ that follows from the completion of the program, following the collect-all and emptiness-as-failure assumptions. We require negated goals and function calls to obey the usual local stratification condition for predicates, as given by [26]. We treat negated goals by negation as failure [19]. When new variables appear in *condition*, i.e., those that are not on the left-hand side of `:-`, then the goals in *condition* should be processed in such an order so that all negated goals and function calls are invoked with ground terms as arguments—because negation-as-failure may in general be unsound for nonground goals.

We illustrate these features by giving two different uses of conditional subset clauses, each of which is motivated by a need to obtain a more declarative or clearer means of solving certain problems than other approaches: (i) Prolog's `setof` feature, and (ii) monotonic aggregation. In discussing (i), we introduce general relational clauses and the use of set terms in them; and in discussing (ii), we introduce partial-order clauses, a generalization of subset clauses.

### *2.2.1. Set terms in Relations and* `Setof` *Operations*

We first provide a couple of examples to illustrate the use of set terms in relational clauses. As noted in the introduction, a relational clause may take one of two forms:

*p(terms)*
*p(terms)* `:-` *condition*

where *terms* and *condition* are as described earlier, except that the set constructor {_/_} is used in defining *terms* instead of the constructor {__}, for a reason which will be apparent from our second example below. Our first example below shows that the definition of set membership can be succinctly stated using a single unit clause:

`member(X,{X/_})`

This predicate can be used to verify set membership, e.g., by `member(b,{a,b,c})`; or to generate the elements of a set one at a time, e.g., by `member(X,{a,b,c})`;

or to insert an element in a set, e.g., by `member(a,S)`. Unlike the list membeship predicate, there is only *one solution* to the latter goal, namely, `S` $\leftarrow$ `{a/_}`. The above `member` predicate may be successively invoked to insert an element into `S`. This example shows that the use of sets helps eliminate unnecessary infinite search trees (as could happen with the list-membership predicate).

Our second example, list-to-set conversion, illustrates further use of the set-constructor {_/_}.

```
list_to_set([ ], phi)
list_to_set([X|L], {X/S}) :- list_to_set(L,S)
```

This predicate can be used to obtain the set representation of a list even if the list has duplicate elements, by a goal such as `list_to_set([1,2,1], S)`. The computed answer for `S` would be {1/{2/{1/phi}}}, which is equivalent to {1,2}. On the other hand, if the {__} constructor were used in the second clause above, the goal `list_to_set([1,2,1], S)` would fail because {1\{2\{1\phi}}} is not a valid term: in any term {X\T}, we require that `X` $\notin$ `T`. For the pair of clauses shown above, there are infinitely many solutions to the goal `list_to_set`(L, {1}, namely, `L` $\leftarrow$ `[1]`, `L` $\leftarrow$ `[1,1]`, `L` $\leftarrow$ `[1,1,1]`, etc. In general, it is preferable to use the {__} constructor for extracting the elements or subsets of a set, and the {_/_} constructor for constructing or inserting elements into a set.

The use of conditional subset clauses to specify `setof` operations was one of the original motivations for introducing this feature [15]. For example, assuming the usual definition of the `append/3` predicate, the Prolog goal

```
|?- setof(pair(X,Y), append(X, Y, [1,2,3]), Answer)
```

for defining different partitions of the list `[1,2,3]` may be expressed as follows:

```
partitions(List) contains {pair(X,Y)} :- append(X, Y, List)
|?- partitions([1,2,3])
```

By the collect-all assumption, the result of the above query is

{pair([ ],[1,2,3])} $\cup$ {pair([1],[2,3])} $\cup$ {pair([1,2],[3])} $\cup$ {pair([1,2,3],[ ])}
= {pair([ ],[1,2,3]), pair([1],[2,3]), pair([1,2],[3]), pair([1,2,3],[ ])}

Note: We extend the emptiness-as-failure assumption so that the result of a query such as `partitions(foo)` is `phi`. That is, when the search tree from the body of a conditional subset clause is finitely failied, the resulting set computed from this clause is the empty set.

The definition of set-difference illustrates use of a negated goal in the body of a subset clause:

```
diff(S1, S2) contains {D} :- member(D, S1), not member(D, S2)
```

The definition of negation-as-failure in this context is slightly more general than the usual definition, in that a goal `not A` will succeed when the search tree for `A` is *finitely failed*, taking into account the branching due to multiple clauses as well as multiple set unifiers within any given clause (due to the set patterns {X/T}).

### *2.2.2. Monotonic Aggregation*

We first note that, since the declarative semantics is stated in terms of *ground instances* of subset clauses, in order to perform a collect-all operation the search

tree emanating from the body of a subset clause must not only be finite but the computed terms must also be ground; otherwise the result of a query is undefined. This requirement is similar in spirit to that of *range restrictedness* in Datalog. For example, consider the clauses

```
f(X) contains {Y} :- p(Y)
p(Y)
```

and a query expression `f(1)`. The answer to this query is undefined because the union of all ground instances of {Y} is an infinite set.

Our first example is a reformulation of the set of reachable nodes in a graph, but now we use the extensional database relation `edge(X,Y)` to represent edges.

```
reach(X) contains {X}
reach(X) contains reach(Y) :- edge(X,Y)
```

The above is a more efficient formulation of the problem because the argument of `reach` (at run-time) will be a constant rather than a set, and hence memo-table lookup can be done more efficiently. Except for this difference, the execution of a top-level query against this program is identical to that of the program given in section 2.1.2.

The following program illustrates the conciseness and clarity of the paradigm for specifying monotonic aggregation. It is a specification of the *company controls* problem [31], and makes use of a *partial-order* clause (for defining `controls`) and two subset clauses (for defining `owns`). The resulting type of the partial-order clause is `boolean`. We can understand this clause in a manner analogous to the subset clause: we simply replace the subset ordering by the boolean ordering `false <= true`, and we replace set-union by the boolean function `or`.

```
controls(X,Y) >= gt(sum(owns(X,Y)), 50)
owns(X,Y) contains {s(X,Y,N)} :- shares(X,Y,N)
owns(X,Y) contains {s(X,Y,N)} :- shares(Z,Y,N), controls(X, Z) = true
```

The function `controls(X,Y)` returns `true` if company `X` controls `Y`, and `false` otherwise. It could be defined in terms of an equational clause just as well; in the above example, the `>=` clause essentially behaves as an equational clause. Note that the function `owns` collects sets of the form {s(X,Y,N)}, rather than {N}, because we wish to sum up duplicate values. The relation `shares(X,Y,N)` means that company `X` holds `N`% of the shares of company `Y`. Cyclic holdings are possible, i.e., company `X` may have direct holdings in company `Y`, and *vice versa*. Therefore, when a circular call occurs on `controls`, the initial lookup value is `false`. Here we see recursion over aggregation: a company `X` controls `Y` if the sum of `X`'s ownership in `Y` together with the ownership in `Y` of all companies `Z` controlled by `X` exceeds 50%. Since percentages are non-negative, `sum` is monotonic with respect to the subset ordering. The function `gt(X,Y)` stands for numeric greater-than, and is monotonic in its first argument with respect to the ordering `false <= true`. Hence the conditions for a well-defined semantics are met. This example also illustrates the use of an aggregate operation, `sum`, which can be defined using equational rules as follows:

```
sum(phi) = 0
sum({s(_,_,N)\T}) = N + sum(T)
```

Our last example further illustrates use of partial-order clauses. Here the resulting domain is `integer` which we take to be the finite set *min_int* ... *max_int* totally ordered under `<=`. In the following program, we make use of a `<=` clause, whose meaning is understood analogous to the subset clause: we simply replace the subset ordering by numeric ordering, and we replace set-union by the numeric function `min2`.

```
short(X,Y) <= C :- edge(X,Y,C)
short(X,Y) <= C+short(Z,Y) :- edge(X,Z,C)
```

The relation `edge(X,Y,C)` means that there is a directed edge from `X` to `Y` with distance `C` which is non-negative. The `+` operator is monotonic, and hence the program is well-defined. Note that, since `short` is circularly defined, the initial lookup value will be *max_int*. (We assume that $x + max\_int = max\_int$). The logic of the shortest-distance problem is concisely and clearly specified in the above program. And our computational model (monotonic memo-tables) provides better efficiency than a dynamic programming algorithm because top-down control avoids solving any unnecessary subproblems. We illustrate this point in section 5. Still, this is *not* the best control strategy for the shortest-distance problem. By specifying that the partial ordering is actually a *total* ordering, it is possible to mimic the Dijkstra shortest-path algorithm. However, our current implementation does not yet support annotations that specify total-ordering.

## 3. Abstract Machine For Subset Logic Programs

An abstract machine is primarily characterized by its instruction set, which in turn refers to global registers and other storage structures for its definition. In this section we present mainly the global registers and storage structures used in the abstract machine; in section 4 we present the instruction set and give examples of compiled code. We present the abstract machine for subset-logic programs in stages: In section 3.1 we briefly review the basic abstract machine for unconditional subset clauses. This description is adapted from [11], which discusses both equational and subset clauses. We then describe in section 3.2 the extension of the basic abstract machine to support memoization and re-do, which are needed in the implementation of unconditional subset clauses. Finally, in section 3.3 we explain the further extensions to handle conditional subset clauses, especially monotonic aggregation.

### *3.1. Basic Abstract Machine*

The overall structure of the basic abstract machine for unconditional subset clauses is depicted in figure 3.1. The main storage areas are the static *code area*, the *control stack* (or local stack), and the *heap*. Unlike the WAM, there is no need for a trail stack here because all function calls will be ground. The control stack is made up essentially of *choice point* and *environment* records, while the heap contains all the data structures constructed during the execution of the body of an unconditional subset clause. A choice point stores information needed for backtracking, whereas an environment record basically holds space for the permanent variables of the clause, and also for the continuation code pointer and continuation environment.

The new feature in a choice point is the set of *branch point* records, which are needed to keep track of the branching within set matching, and are explained below.

- *Choice and Branch Points.* The multiple subset clauses that match a given call and the multiple set matches within a single subset clause are attempted sequentially in depth-first order. We create a *choice point* record on the control stack to keep track of these alternatives; a choice point is needed even if a function is defined by a single subset clause, provided there is at least one set constructor at the head of this clause. This choice point serves two purposes: it will preserve the environment for call-one invocation and it will protect the environment for a *redo*, as we will see in the next subsection. In the case of multiple subset clauses, as in the WAM a choice point is created below the environment record of all subset clauses. If any of these subset clauses has a set constructor, an additional choice point would be created above the environment record of this subset clause, as explained above. In order to keep track of the state of the match for set terms of the form `{H1\T1}`, we augment the abstract machine with a set of *branch registers*, one for each distinct occurrence of the set constructor or the left-hand side of a clause. The branch register points to the *branch point record* which records the address of the instruction (see section 4) as well as the current bindings of `H1` and `T1`, since new matches for `H1` and `T1` are constructed from previous matches (see [11] for details). We think of branch point records as extending the choice point record of the subset clause; that is, a single choice point can contain multiple *branch points*. A choice point holds the number of arguments, their actual values, a pointer to its current environment (`CE`), a continuation pointer (`CP`), and a pointer to the last choice point (`MRCP`), and the number of branch points (`BP`). Except for `BP`, the other registers are similar to those of the WAM. We will see later how this contents are extended further.

- *Environment Records.* The creation of an environment record for a subset clause depends upon the calling mode, which is stored in a special register called the *mode register*: In the *call-all* mode, an environment record is created if a subset clause matching this call has at least one call in its body. In the *call-one* mode, all variables are assumed to be permanent whether or not function calls are present in the body of the subset clause. This is done since a call-one invocation returns one element of a set at the time, and it needs the state of the computation to be preserved for later resumption. An environment record holds the number of permanent variables, the values of these variables, a pointer to its continuation environment (`CE`), and a continuation pointer (`CP`). When control reaches the end of a subset clause, the environment record is deleted provided all branch points within this clause have been explored and the current environment record is at the top of the control stack.

- *Success Backtracking.* Note that failure cannot occur with unconditional subset clauses—the emptiness-as-failure assumption states that, if a function call fails to match the head of any clause, the result of the call is the empty set, `phi`. Success-backtracking depends upon the calling mode: For a call-all invocation of a subset clause, each time control reaches the end of the clause,

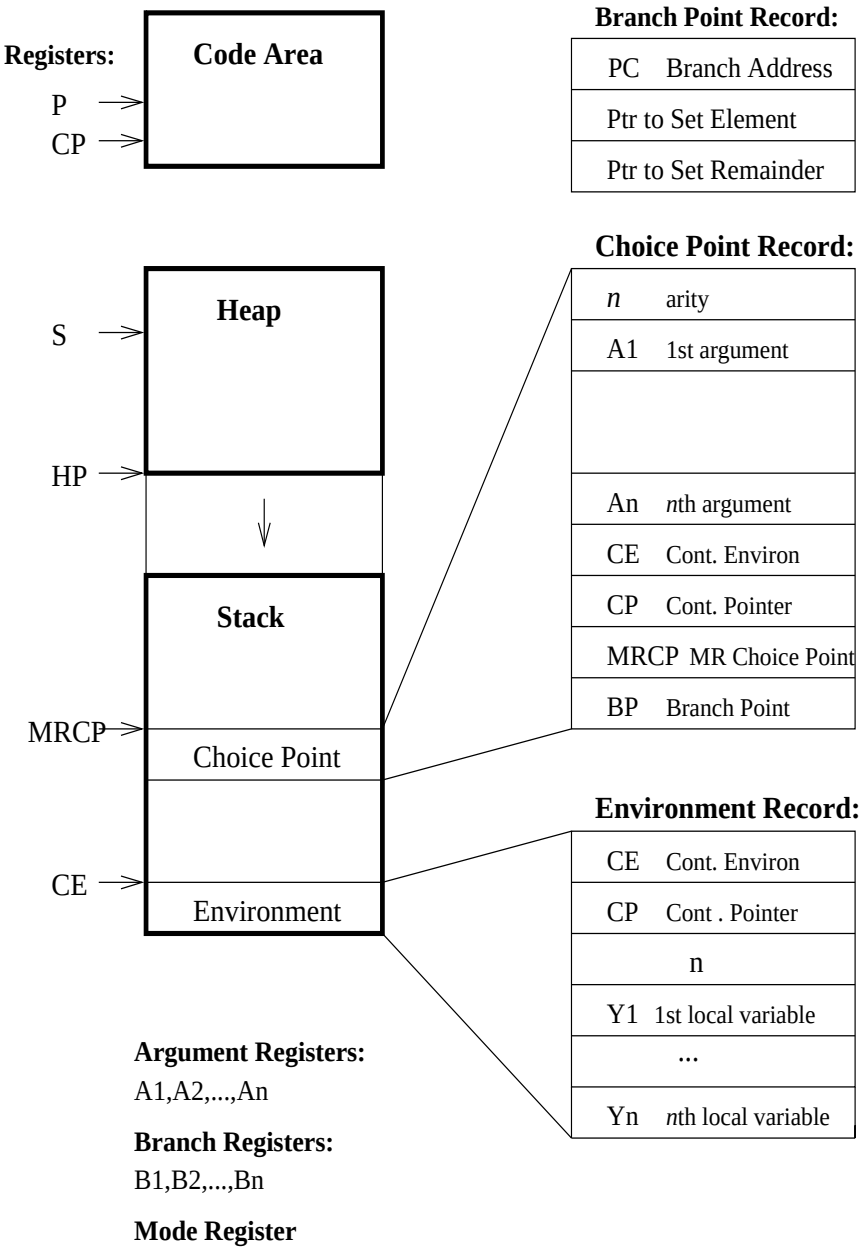


Figure 3.1: Basic Abstract Machine

success-backtracking to the most recent choice-point occurs. As each subset is computed, it is added to the overall set after removing duplicates. For a call-one invocation of a subset clause, each time control reaches the end of the clause, an exit occurs back to the caller before generating the next element.

## *3.2. Abstract Machine for Memoization and Re-Do*

Before describing the run-time structures needed to implement memoization and re-do, we first describe more precisely the computation of a goal involving these two operations. We assume that any function defined recursively through subset clauses will be memoized. This is a safe strategy in the sense that it will detect any circular function call that may arise. A memoized call is said to be made in *call-memo* mode, as opposed to call-all or call-one mode.

### *3.2.1 Abstract Computational Model*

We illustrate the operation of the abstract computational model by means of a very simple example.

```
g(X) contains {10}          h(X) contains {20}
g(X) contains h(X)          h(X) contains p(g(X))

p({X\_}) contains {X,30}
```

We *flatten* the body expression of a subset clause and replace it by a sequence of equality goals; the order of goals reflects a leftmost-innermost calling sequence. For example, an expression `p(g(X))`, where `g` and `p` are non-constructor (i.e., user-defined) function symbols, is flattened as
`g(X) = T1, p(T1) = S2`.
Similarly, a top-level query such as `g(1)` will be flattened as `g(1) = Ans`. Note that the memo-table records with each memoized call the value `phi` as the initial approximation of its result. (For functions that are defined by general partial-order clauses, the initial value depends upon the type of the result and the kind of partial order clause, `<=` or `=>`). For the above program, the derivation from the query `g(1)` is as follows.

| **Goal Sequence** | **Substitution** | **Memo Table** |
|---|---|---|
| `g(1)=Ans` | `Ans` $\leftarrow$ {10} $\cup$ `S1` | $\phi$ |
| `h(1)=S1` | `S1` $\leftarrow$ {20} $\cup$ `S2` | {`g(1)=phi`} |
| (L) `g(1)=T1, p(T1)=S2` | `T1` $\leftarrow$ `phi` | {`g(1)=phi, h(1)=phi`} |
| `p(phi)=S2` | `S2` $\leftarrow$ `phi` | {`g(1)=phi, h(1)=phi`} |
| (R) `g(1)=T1, p(T1)=S2` | `T1` $\leftarrow$ {10,20} | {`g(1)={10,20}, h(1)={20}`} |
| `p({10,20})=S2` | `S2` $\leftarrow$ {10,20,30} | {`g(1)={10,20,30}, h(1)={10,20,30}`} |

| | | |
|---|---|---|
| (R) g(1)=T1, p(T1)=S2 | T1 ← {10,20,30} | {g(1)={10,20,30}, h(1)={10,20,30}} |
| p({10,20,30})=S2 | S2 ← {10,20,30} | {g(1)={10,20,30}, h(1)={10,20,30}} |
| [] | | {g(1)={10,20,30}, h(1)={10,20,30}} |

A memo-table lookup occurs in step 3 and is indicated in the above derivation by a (L) against the goal `g(1) = T1`. The value retrieved from the table is `phi`. Using this value, the computed answer for `h(1)` is {20} and for the top-level query `g(1)` is {10,20}. The memo-table is updated with these values as soon as they are computed. Since the computed value for `g(1)` at the top-level is different from that assumed for this call in step 3, the call `g(1)` is subjected to a re-do, and this is shown by a (R) against the goal `g(1) = T1` in the step 5. The revised value retrieved from the table is {10,20}. Using this value, the revised computed answer for the top-level query is {10,20,30}. Once again a re-do is required (see step 7), and this time the computed answer for the top-level query agrees with the value retrieved from the memo-table, and hence the computation terminates with {10,20,30} as the final answer.

The above program raises the issue of how often a re-do should be performed. For example, in the case of function `g` above, we can, in principle, we can perform a re-do after *each* clause for `g` has completed, rather that perform the re-do after *all* clauses for `g` have completed. We have experimented with both strategies in our implementation. From these experiments, we have found that performing the re-do after every clause involves more computational steps (since a body code is being executed many more times) and hence we favor performing this operation after all clauses have completed.

### *3.2.2 Lookup Points*

It is clear that the basic abstract machine outlined in section 3.1 must be extended with a memo-table in order to execute the full language of unconditional subset clauses. However the control must be suitably modified in order to correctly implement the re-do process. We illustrate the problem and its solution by reconsidering the above derivation. Figure 3.2a shows the state of the control stack at step 4 of the above derivation. Note that a lookup occurred when the environment record for `h(1)` was on top of the stack. After the call to `p(phi)` is completed, under normal execution the environments for `h` and `g` would be deallocated, and we will not be able to re-do the goal `g(1)=T` of step 3 of the above derivation.

To solve the above problem we push a new record, called a *lookup point*, on the control stack above the environment record wherein a lookup occurred. The lookup point preserves the state of computation at the time when a memo-table lookup occurred, so that all the necessary information can be retrieved upon a re-do. All lookup points are chained together, and the head of this list is kept in a new global register called the *most recent lookup point* register (`MRLP`). A re-do operation is initiated whenever a memo-table entry is updated and there was lookup made on

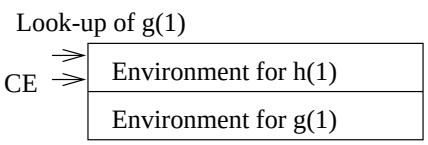


Figure 3.2a

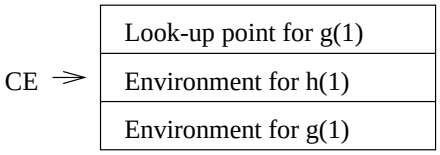


Figure 3.2b

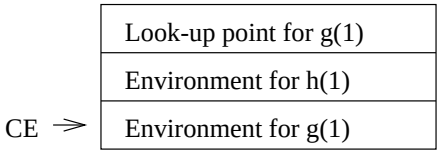


Figure 3.2c

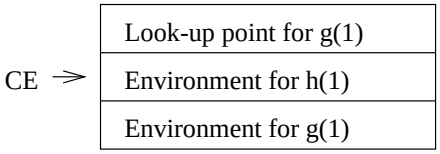


Figure 3.2d

Figure 2: Illustrating the Need for Lookup Points

this entry, i.e., there is a lookup point corresponding to this entry on the control stack. In general if there is more than one lookup point on the control stack, all lookup points that are more recent than the one that was responsible for the re-do are discarded; they will be re-created as needed during re-execution. Thus the information in a lookup point consists of the most recent lookup point (`MRLP`), the continuation point (`CP`), the current environment (`CE`), the most recent choice point (`MRCP`), and the location to return the lookup value.

Returing to the above derivation, figure 3.2b shows the control stack after step 4 of the above derivation, assuming a lookup point has been created. After the call `p(phi)` is completed, environments for `h(1)` and `g(1)` will not be deallocated because they are protected by a lookup point for `g(1)`. When the call `h(1)` completes, the memo-table entry for `h(1)` is updated to {`20`}; and when the call `g(1)` completes, its memo-table entry is updated to {`10,20`}. Figure 3.2c shows the state of the control stack at this point: The current environment pointer (`CE`) points to the bottom of the control stack, but the environments for `g(1)` and `h(1)` are not deleted. When the memo-table entry for `g(1)` is updated to {`10,20`}, a re-do is initiated. This results in `CE` being set to the environment for `h(1)`. The resulting control stack is shown in figure 3.2d.

To summarize the discussion, the basic operations related to memoization are: creation of a memo-table entry, updating the memo-table entry, memo-table lookup, and re-do. The overall structure of the abstract machine for unconditional subset clauses is shown in figure 3.3. To facilitate efficient access to the memo-table, it is organized as hash-table whose efficiency is well-known for dictionary type operations. To facilitate efficient re-do, we record with each memo-table entry a pointer to the most recent lookup point that consulted this value (this pointer is `nil` if there was no lookup made on this entry).

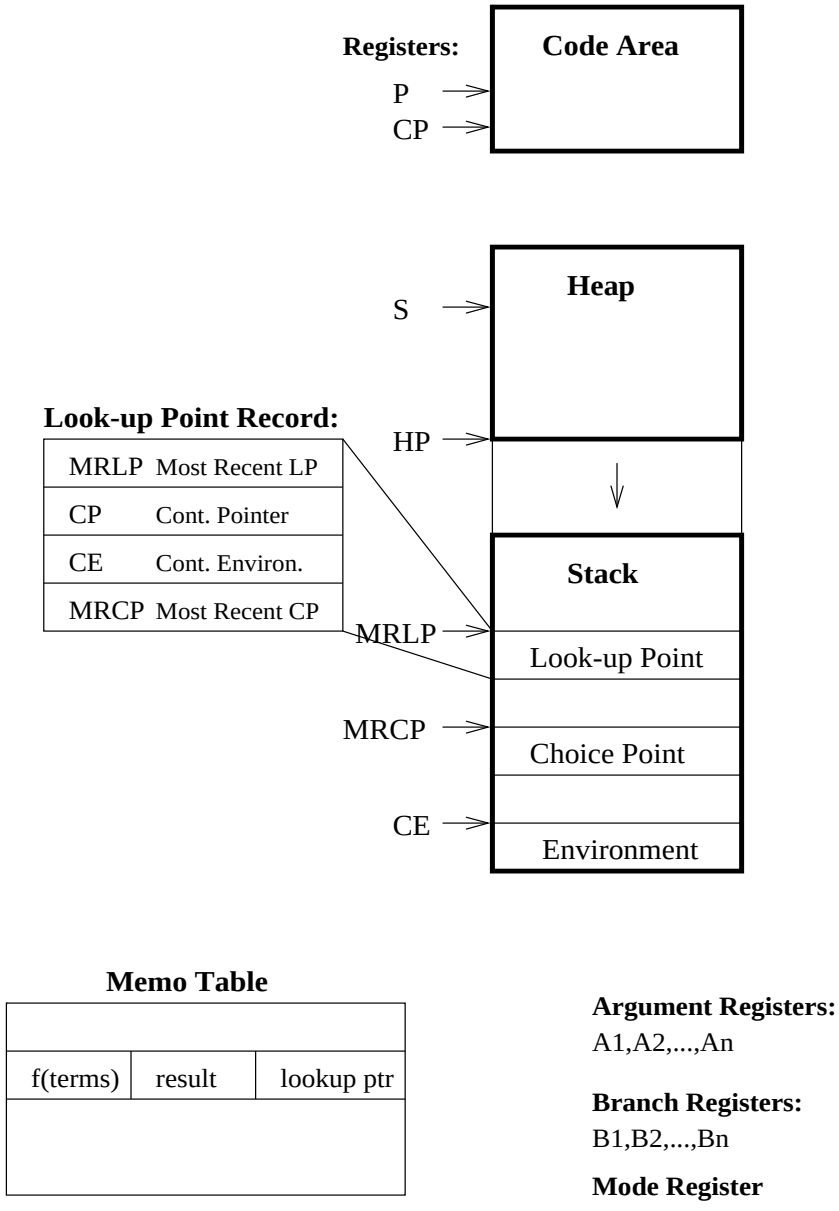


Figure 3.3: Abstract Machine for Memoization and Re-Do

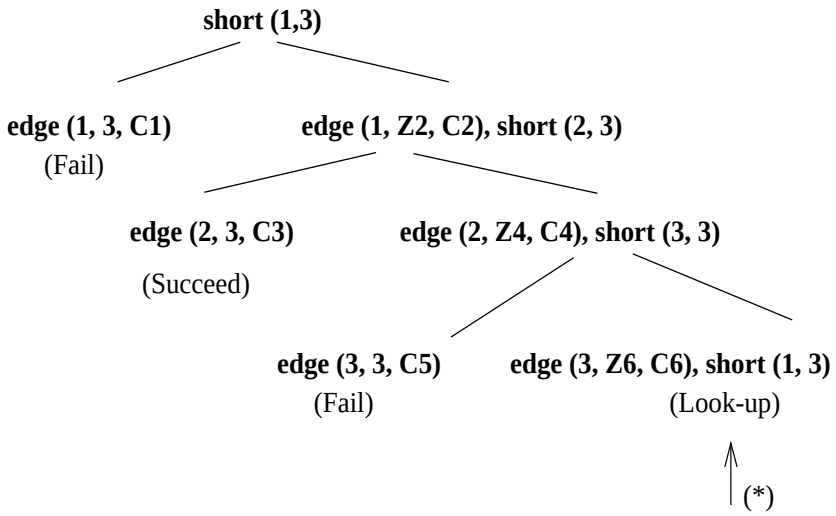


Figure 3.4: Illustrating the Need for Re-do Trail

### *3.3. Abstract Machine for Conditional Subset Clauses*

We develop the abstract machine for conditional subset clauses by extending the machine for unconditional subset clauses to handle relational clauses and goals. When relational clauses are present, we need to maintain a trail stack for resetting variables upon backtracking. In addition, we need to extend the control mechanism so that re-do works correctly in the presence of relational goals in the bodies of conditional subset clauses, e.g., for programs involving monotonic aggregation.

To illustrate the implementation problem involving conditional subset clauses with relational goals in their bodies, consider the `short` example from section 2.2.2 along with a few instances of the `edge` predicate:

```
short(X,Y) <= C :- edge(X,Y,C).
short(X,Y) <= C+short(Z,Y)  :- edge(X,Z,C).
edge(1,2,10).
edge(2,3,20).
edge(3 1,30).
edge(3,4,40).
```

The call tree of the goal `short(1,3)` is shown in figure 3.4. Note that there is a cyclic call on `short(1,3)` and therefore a lookup occurs on this goal. After the completion of this lookup, control returns to the point marked (*), since there is a choice point created as a result of the goal `edge(3,Z6,C6)`. Now the variables `C6` and `Z6` are reset, and the computation proceeds by determining new bindings for these two variables (`Z6=4`, `C6=40`) and calling `short(4,3)`, which results in no further calls on `short`. When control eventually returns to the top-level, the memo-table entry for `short(1,3)` is updated, and a re-do on the cyclic call is initiated. However, since the bindings for `C6` and `Z6` were reset, the environment in which this call is to be re-done must be restored to its original condition.

To solve the above problem, at the time a lookup is performed, if the `MRCP` is positioned above the lookup point, we must save the bindings of those variables

belonging to the current or any more recent environment that were trailed. This calls for a new storage structure which we refer to as the *redo-trail*. The XSB system also makes use of a similar data-structure in its implementation. In the above example, the bindings of variables `Z6` and `C6` are saved in the redo-trail at the time of lookup, and they are recovered at the time of re-do. Thus, the lookup point now maintains a pointer into the redo-trail to facilitate the recovery of information during a re-do.

The overall abstract machine for conditional subset clauses is summarized in figure 3.6.

## 4. Instruction Set for Subset Logic Programs

The execution of SuRE programs involves two steps: compilation, followed by the interpretation of the compiled code. The basic scheme for the implementation for SuRE follows that of the WAM for Prolog: Our adaptation of the WAM supports the compilation of set-matching, set-unification, as well as memo functions. Furthermore, a more complicated control cycle than that of Prolog is necessitated by the subset clause, due to the ability to execute set-valued functions in different modes—call-one call-all, and call-memo—as well as the need to re-execute function calls (re-do). The SuRE compiler was written in Quintus Prolog and it generates "SLAM code" (Subset-Logic Abstract Machine code). This code is emulated by a run-time system which was written in C. Below we describe the main steps in compilation.

This section explains each category of instructions, starting with instruction set for unconditional subset clauses, followed by the additional instructions needed for a full language. We describe the instruction set in stages, with emphasis on the differences from those for Prolog. Section 4.1 presents the instruction set for the basic abstract machine of section 3.1; section 4.2 presents the additional instructions for unconditional subset clauses with memoing; and, finally, section 4.3 presents the instructions for the full language, especially set unification.

### *4.1. Instruction Set for the Basic Abstract Machine*

The basic abstract machine implements the sublanguage of unconditional subset clauses without memoization. Only functions (i.e., no predicates) are permitted in this sublanguage. We compile a function with $n$ arguments as a predicate with $n+1$ arguments, the last argument standing for the result of the function. Of course, we assume that the first $n$ arguments are invoked with ground arguments. Overall, there are seven classes of instructions—*get*, *put*, *match*, *store*, *store_indirect*, *indexing*, and *procedural* instructions. Of these, the *put*, *store*[2], and *procedural* instructions are similar to those of Prolog. The remaining instructions are as follows:

*Get Instructions and Match vs. Store_Indirect Instructions*
Get instructions are used to compile the argument occurring in the head of a clause. Since every argument to a function call is ground, we can identify the *read* and *write* modes of WAM's *get* instructions at compile-time. Thus, unification instructions

[2]This instruction set is similar to the set instructions given in [1], which correspond to unify instructions in 'write' mode.

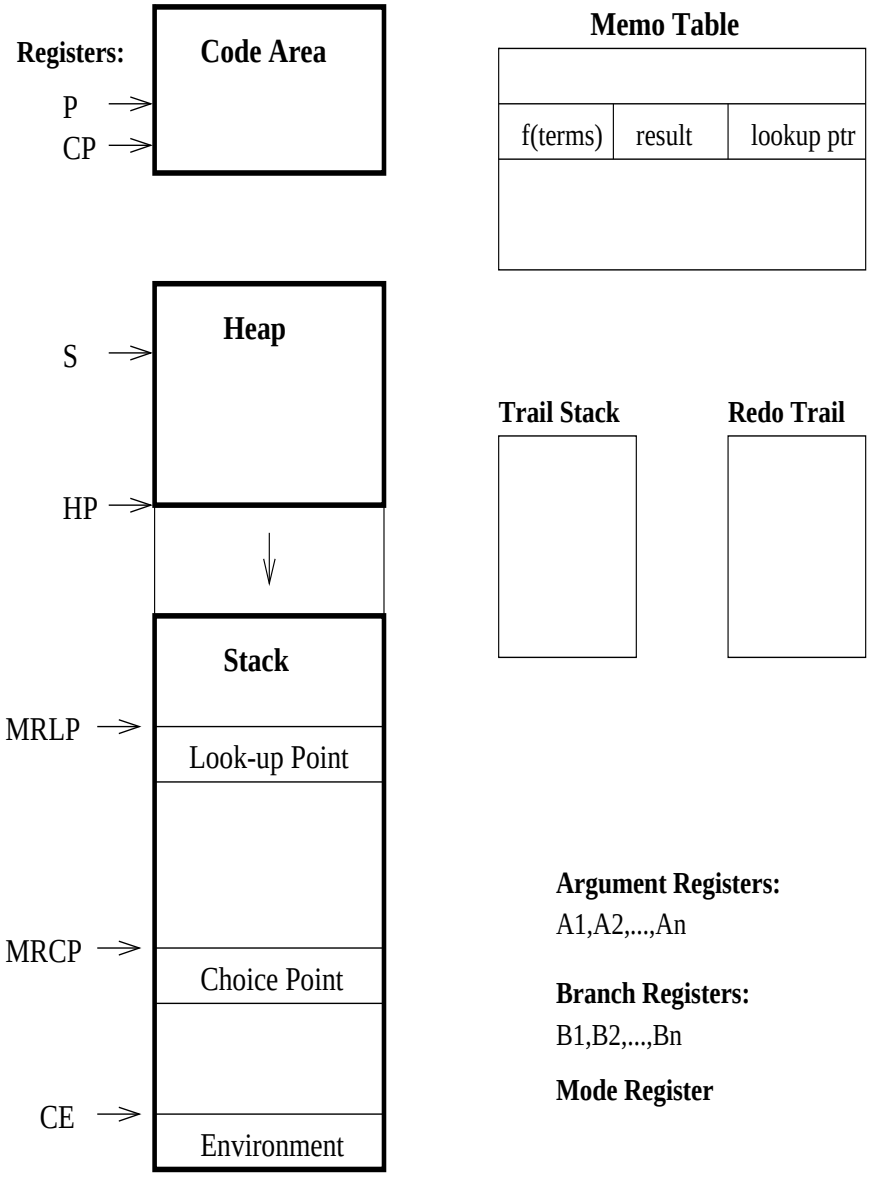


Figure 3.5: Abstract Machine for Subset-Logic Programs

in read mode will be compiled into *match* instructions, i.e. one-sided unification, and those in write mode will be compiled into *store_indirect* instruction, to build a term. Except for a set term, every structured term occurring in the head will be compiled into a get instruction followed by match instruction.

*Set-Matching Instructions*
In order to compile the set constructors of the form, `{_\_}`, appearing at the head of a subset clause, we introduced in [11] three instructions—`adj_set_head`, `adj_set` and `adj_set_with_copy`—each of which treats the remainder-set in a different way. Basically these instructions are used immediately after a `get_set` instruction, to "adjust" the matching set so as to prepare it for the next match [11]. These three instructions cannot be merged with the preceding `get_set` instruction to give three new `get_set` instructions because they may be the target of backtracking, whereas the `get_set` is not. (Each *get* instruction either binds the argument register to temporary or permanent register or dereference its contents). The `adj_set_head` instruction is used when the set term has the form `{X\_}`, meaning that no remainder set need be constructed; matching can proceed by effectively binding `X` in turn to the individual elements of the matching set. The `adj_set` instruction is used for set terms of the form `{X\T}`, where each new remainder set for `T` can be constructed from the previous one for `T` by destructive modification. This instruction can be used if the set bound to variable `T` is not being returned, either directly or indirectly, as part of the function's result. Otherwise the `adj_set_with_copy` instruction must be used, meaning that each new remainder set cannot be formed by destructive modification from the previous one.

*Procedural Instructions*
The procedural instructions of the WAM are augmented with `call_one`, `call_all`, `last_call_one` and `collect` instructions. The `call` instruction of the WAM is replaced by the `call_one` or `call_all` instruction depending on the calling mode. The `execute` instruction of the WAM is supplemented with a `last_call_one` instruction whose role is similar to those of `call_one` and `call_all` instructions, except that it is used to compile the last call. In [11] the `last_call_one` instruction is used when the last-call register `LC` indicates that last-call optimization (LCO) [35] is possible. For reasons discussed in the previous section, however, LCO is no longer possible if a goal in the body is a memo function; the environment needs to be protected for a redo if there is a lookup point above it. The compiled code for each subset clause ends with a `collect` instruction whose behavior depends on the calling mode (given in the mode register): In the call-all mode, it is responsible for constructing the resulting set, removing duplicates and also for eventual deletion of the environment record; in the call-one, it simply exits back to the caller. Note that, in the call-one mode, the `deallocate` instruction will not deallocate the environment if there is a choice point above it, but it will in the call-all mode.

*Index Instructions*
As in the WAM, a program clause is indexed on its first argument. However, because every functional argument is ground, we use `switch_on_ground_term`, a new instruction with four cases: constant, list, structure, and set. We use the instruction `try_sub_and` when the left-hand sides of two or more subset clauses can potentially match a given function call. This instruction creates a choice-point

frame. A `save_choice_point` instruction is placed after the compiled instructions for the left-hand side and before the compiled instructions of the body of each subset clause. It also creates choice points if there is a branch point due to different set matchings.

For a complete example, consider the following subset clauses for `perms`:

```
perms(phi)    contains {[]}
perms({X\T}) contains distr(X, perms(T))
```

The compiled code should be self-explanatory with the comments.

```
perms/1: switch_on_ground_term   C1, fail, fail, C2
C1:      allocate                2           %
         get_phi                 A1          % phi)
         save_choice_point       2           % contains
         store_indirect_var      Y1, A2      %
         put_set                 Y2          % {
         store_nil                           % []
         store_phi                           % }
         collect                 Y1, Y2      %
C2:      allocate                6           %
         get_set                 Y6, A1      %  {x\t}
         adj_set_with_copy       Y6          %
         match_variable          Y1          %
         match_variable          Y2          %  )
         save_choice_point       2           %  contains
         store_indirect_var      Y3, A2      %  v :-
         put_value               Y2, A1      %  perms(t) =
         put_variable            Y4, A2      %  v1,
         call_all                perms/1     %
         put_value               Y1, A1      %
         put_value               Y4, A2      %  distr(s, v1) =
         put_variable            Y5, A3      %
         last_call_one           distr/2     %
         collect                 Y3, Y5      %  v.
```

## *4.2. Instruction Set for Memoization*

We introduce three new instructions, `call_memo`, `execute_memo`, and `update_memo`, to implement memoization. Both `call_memo` and `execute_memo` instructions create a memo-table entry if the function is being called for the first time on these arguments. and they perform a lookup if it is not the first time. The `execute_memo` instruction is for the last call. Both instructions are called in call-all mode since a memo-table is updated after its call is completed. The `update_memo` instruction is to update a memo-table entry for a memo function call when it is completed and triggers a redo if necessary. Therefore, it follows immediately either a `call_memo` or an `execute_memo` instruction. Finally, the operation of the `allocate` instruction needs to be changed since we have introduced a new global register `MRLP` for a lookup point. Now, any new environment, choice point or lookup point will be created above `CE`, `MRCP`, and `MRLP`.

We illustrate the compiled code for the function `reach` of section 2.1.2.

```
reach(V) contains {V}
reach(V) contains reach(W) :- edge(V, W)
```

Note that the second clause of the `reach/1` definition is flattened and transformed[3] as follows:

```
reach(V) contains S1 :- edge(V, W), all(W) = S1
all(W) contains reach(W)
```

The compiled code is as follows, and should be self-explanatory in view of the comments accompanying each instruction. The `execute_memo` instruction performs a memoized call and the `update_memo` instruction updates a memo-table entry.

```
reach/1: try_sub_and            L1
         allocate               3
         get_variable           Y1, A1          % reach(V)
         save_choice_point
         store_indirect_var     Y2, A2          % contains
         put_set                Y3              % {
         store_value            Y1              % V
         store_phi                              % }
         collect                Y2, Y3          % ∪ V
L1:      allocate               4
         get_variable           Y1, A1          % reach(V)
         save_choice_point                      % contains
         store_indirect_var     Y2, A2          % S1 :-
         put_value              Y1, A1          % edge(V,
         put_variable           Y3, A2          % W),
         call                   edge/2
         put_value              Y3, A1          % all(W)
         put_variable           Y4, A2          % = S2
         execute                all/1
         collect                Y2, Y4          % S1 := S1 ∪ S2
all:     allocate               3
         get_variable           Y1, A1          % all(W)
         save_choice_point                      % contains
         store_indirect_var     Y2, A2          % S1 :-
         put_value              Y1, A1          % reach(W)
         put_variable           Y3, A2          % = S2
         execute_memo           reach/1
         update_memo            reach, 1, Y3
         collect                Y2, Y3          % S1 := S1 ∪ S2
```

### *4.3. Instruction Set for the SuRE Language*

The new instructions for the full subset-logic language primarily pertain to the procedural and unification classes of the WAM. (While there are additional instructions

[3] This transformation is done in order to enforce set-collection and is discussed in more detail in [11, 20].

pertaining to lazy evaluation and set enumeration goals, we do not discuss them in this paper.) To link together two or more relational clauses, we use the WAM instruction `switch_on_term` (except that its argument can also be set), along with the `try_me_else` and `trust_me` instructions. The rest of this section is devoted to the instructions for set unification.

The *unify* instruction set is similar to that of the WAM. Terms appearing in the head of subset clauses will be compiled as before, while terms appearing at the head of a relational clause, excepting set terms, will be compiled as in Prolog. All equational goals will be compiled assuming that their arguments are ground—our compiler performs groundness checks and will flag violations—while relational goals will be compiled as in Prolog. The presence of set terms of the form `{_/_}` in relational clauses requires new instructions for set unification because [11] only introduced instruction sets for *set matching*, adjusting an incoming set combined with `get_set` instruction.

Since it cannot be assumed that an incoming term would be ground for relational clauses, adjust-set instructions cannot be used. Since set unification generates multiple unifiers, the `get_set` instruction alone is not enough. Furthermore, unlike set matching where only an incoming set is being adjusted, set unification requires adjusting of both terms and needs to know both terms before applying unification. Thus, the usual compilation of structured terms will not suffice. We therefore create several new instructions so that a set argument appearing in the head of a relational clause can be built on the heap. We then apply the set-unification algorithm on these terms (incoming argument and the argument just built). New instructions for set unification are discussed below.

*The w_get instructions:* There are four *w_get* instructions: (1) `w_get_variable` $Y_n, A_i$, (2) `w_get_structure F,` $Y_i$, (3) `w_get_list` $Y_i$ and (4) `w_get_set` $Y_i$. They are essentially *get* instructions in the 'write' mode. The `w_get_set` instruction would treat a set like any other structured term except that it builds a set. Any structured term appearing inside of a set term will be compiled with these instructions; the mode will always be 'write'.
*The write instructions:* These are: (1) `write_variable` $V_n$, (2) `write_value` $V_n$, (3) `write_constant C`, (4) `write_nil`, and (5) `write_phi`. These instructions correspond to *unify* instructions in 'write' mode, i.e., they will build a term on a heap. Terms that would be compiled into unify instructions will be compiled with these if they appear inside of a set term in the head instead.
*The unify_set instruction:* This instruction performs set-unification on two terms, generates a unifier, and is responsible for creating branch points. If set unification generates more than one unifier it will create branch points as in set-matching. Like set matching, it needs to try one unifier at a time, and backtracks if necessary. Backtracking by branch points causes control to return to the branching instruction—`unify_set` in this case.

In addition, since a set term appearing in the head of a relational clause may introduce branch points and it may be a target of a redo operation as seen in the previous section, we place a `save_choice_point` instruction at the end of the compiled code of each relational clause that has a set term in its head. Below we will show the operations for the new instruction set and illustrate their use with an example. Some similarity between the `unify_set` instruction and the adjusting-set instructions can be noted—both are targets of backtracking. But, unlike the adjust-set

instruction, which simply sets a pointer for the next match, the `unify_set` instruction is responsible for binding variables for the next unifier. Other instructions that call a unification routine can also be the target of backtracking, e.g., `get_value` and `unify_value`, when the terms they unify are set terms.

We illustrate below the compiled code for a relational clause with a set term.
`member(X, {X/_})`

```
member/2: allocate            2
          get_variable        Y1, A1          % member(X,
          w_get_set           Y2              % {
          write_value         Y1              % X/_}
          unify_set           Y2, A2
          save_choice_point   2
          deallocate
          proceed
```

## 5. Experimental Results

In this section we discuss the results of a prototype implementation of the foregoing ideas and techniques, and compare this implementation with two closely related systems. We completed this implementation in the summer of 1996, and all of the examples shown in this paper were tested out using this implementation. Section 5.1 will describe performance figures of SuRE programs, concentrating on the performance of memoization and re-do, as these are the main new features of our implementation. Section 5.2 will show performance comparisons on representative programs with XSB and CORAL systems.

### *5.1. Performance Figures for SuRE*

We note at the outset that the memo-table is organized as a hash-table (with chaining), the cost of a lookup and creating a memo-table entry is a constant independent of the size of the table as long as a load factor $\alpha < 1$. The load factor is the average number of elements stored in a chain. We assume that the size of a memo-table is large enough and the hash function is fairly uniform. This means there is at most one entry on the chain on average. Our current implementation actually does not guarantee that the load factor is always less than one. To ensure this condition, we should be able to increase the hashtable periodically. For example, we can set a certain threshold, and if the number of entries exceeds it, the hash-table size is increased (e.g., doubled). In our prototype implementation, a hash-table with 4096 entries was used. However, this size can easily be increased if one were interested in experimenting with larger problem sizes.

When the arguments to a memoized function are structured types, such as a list or set, it will take $\theta(n)$ to form a key, where $n$ is the size of argument list. An important point to note about memo-table updating is that, since the environment for a memoized call will remain on the control stack as long as its computation is not completed, keeping a pointer to the memo-table entry for the call allows direct access to the entry and hence updating would be a constant time operation. However, the check for a re-do can be costly if the result is a large set—the new

value must be compared with the one in the table in order to decide whether a re-do is needed.

In the next two subsections we report performance figures from running several of the programs. We first discuss the performance of memoization and then the performance of memoization with re-do. All timings shown represent in *seconds* the execution times on a Sun Ultra Enterprise 4000, with 168 MHz UltraSPARC CPUs and 1 gigabyte of main memory. Each timing shown for a particular input and program is actually the average of several runs of the program on this input. Care has been taken so that the memo-table is purged in between runs.

*Performance of Memoization*

First we consider an example where memoization is used purely for efficiency reasons. The function `short1` below is a direct rendering of the dynamic programming formulation of the shortest-distance example. Here, the third argument of `short1` specifies a bound on the number of edges to be traversed in finding the shortest distance and serves to explicitly identify subproblems. Since no cyclic calls will arise in this program, it does not require re-do for its correct execution.

```
short1(X,Y,1) <= C :- edge(X,Y,C)
short1(X,Y,L) <= short1(X,Y,L-1) :- L > 1
short1(X,Y,L) <= C+short1(Z,Y,L-1) :- L > 1, edge(X,Z,C)
```

The above formulation can be contrasted with the formulation shown in section 2.2.2, which we refer to further below as `short2`. The running times for the above program for increasingly larger graphs is shown below. The number in parentheses next to each performance figure refers to the value of `L` that was used in making the top-level call (we use the smallest value needed to compute the correct answer). Without memoization, the computational cost grows exponentially in the number of nodes, whereas, with memoization, this cost grows polynomially with the number of nodes.

| | *# of nodes (# of edges)* | | |
|---|---|---|---|
| | 8(18) | 16(38) | 25(58) |
| *with memo* | 0.0032(2) | 0.0303(6) | 0.09799(10) |
| *without memo* | 0.0027(2) | 0.6398(6) | 155.274(10) |

Our next example is the function `reach` which performs a transitive closure operation.

```
reach(X) contains {X}
reach(X) contains reach(Y) :- edge(X,Y)
```

As noted in section 2.2.2, this function does not require a re-do operation even for a cyclic input graph (such a case can be detected syntactically during compilation). It requires only a lookup which is a constant-time operation since the argument to `reach` is a simple constant. However, the result of `reach(X)` is a set, and the cost of constructing it affects the overall performance.

| | *# of edges* | | | |
|---|---|---|---|---|
| | 25 | 50 | 100 | 200 |
| *graph1* | 0.0061 | 0.0120 | 0.0239 | 0.0482 |
| *graph2* | 0.0217 | 0.0750 | 0.2957 | 1.0415 |

In the preceding table, *graph1* is a balanced binary tree (an acyclic graph), and therefore does not require memoization for correct execution. Still, every call on `reach` is memoized by the implementation since it cannot detect this fact from the extensional database of `edge` facts. Nevertheless the performance grows linearly with the number of nodes in the graph, showing that memoization does not degrade the overall performance.

In contrast with *graph1*, all the nodes of *graph2* are arranged in a single cycle. Every call to `reach` is once again memoized. Since, for each `X`, `reach(X)` is the *entire* set $N$ of nodes in the graph, the size of each set in the memo-table will be $|N|$. The cost of duplicates-checking incurred in forming these sets makes the overall performance for *graph2* quadratic in the number of nodes. In contrast, in the case of *graph1*, the average size of each set in the memo-table will be $log_2(|N|)$.

Next we show an example where the benefits of memoization are lessened when the function being memoized is set-valued and its argument terms are also sets. The function `parts` below computes the set of all partitions of a set into two disjoint subsets.

```
parts(phi) contains {pair(phi,phi)}
parts({H\T}) contains distr2(H,parts(T))
distr2(H,{pair(P1,P2)\_}) contains {pair({H/P1},P2), pair(P1,{H/P2})}
```

The size of the output set is $2^n$ where $n$ is the size of input set. Any call on `parts` with an argument set of size $n$, where $n \geq 2$, will result in $n \times (n-1)/2$ identical pairs of calls on `parts` with argument sets of size $n-2$. Hence there is ample opportunity for memoization to speed up the computation here: we expect noticeable improvements as the input set becomes larger. However, the cost of set-equality checks in the memo-table greatly diminish the benefits of avoidance of redundant computations.

| | *Size of argument set* | | | |
|---|---|---|---|---|
| | 4 | 5 | 6 | 7 |
| *with memo* | 0.0102 | 0.0663 | 0.4389 | 13.59 |
| *without memo* | 0.0115 | 0.1006 | 0.6157 | 15.36 |

Finally, we report on the performance of SuRE programs when a bag is constructed instead of a set, i.e., when duplicates-checking is explicitly suppressed. Bag construction is illustrated by the program below. The annotation `bag perms/1` indicates that duplicates-checking need not be performed on the resulting set computed by `perms`.

```
bag perms/1
perms(phi)  contains  {[ ]}
perms({X\T})  contains  distr(X, perms(T))
distr(X,{L\_}) contains  {[X|L]}
```

The following table shows the results for the case when the bag annotation is present and when it is absent (i.e., a set is constructed as the result).

| | *Size of input set* | | | |
|---|---|---|---|---|
| | 4 | 5 | 6 | 7 |
| *with bags* | 0.0115 | 0.1006 | 0.6157 | 15.36 |
| *with sets* | 0.0149 | 0.1592 | 3.185 | - |

In both cases, the function `perms` was called in the call-all mode. (Note that, due to the prohibitive space and time requirements, it was not possible to obtain the figures for an input set of size 7 when working with sets.) The table below shows the performance of `perms` in the call-one mode. We conclude that incremental computation with bags is the most efficient scheme whenever it is applicable.

| | *Size of argument set* | | | |
|---|---|---|---|---|
| | 4 | 5 | 6 | 7 |
| *with bags* | 0.0127 | 0.1046 | 0.6083 | 10.92 |
| *wit sets* | 0.0156 | 0.1496 | 2.844 | 127.8 |

*Performance of Re-Do*

We present two different examples to illustrate the performance of re-do. First we compare the behavior of the formulation of the shortest-distance example with and without re-do. The function `short1` shown earlier requires memoization but not re-do. The function `short2` below requires both memoization and re-do. One important advantage of `short2` over `short1` is that `short2` does need the third argument of `short1`. That is, `short2` does not need to have knowledge of the bound on the number of edges to be traversed in finding the shortest distance.

```
short2(X,Y) <= C :- edge(X,Y,C)
short2(X,Y) <= C+short2(Z,Y) :- edge(X,Z,C)
```

We compared the performance of these two functions on the same graph (represented by the `edge/3` predicate). The table below shows that `short2` outperforms `short1` as the graph size increases. The graph below has no regular property, but has several cycles in order to exercise the re-do process. The reason for the better performance of `short2` is that it avoids solving unnecessary subproblems. The numbers in parentheses in the row for `short2` refer to number of redo's that were performed. These figures indicate to us that re-do is a reasonably efficient means of performing monotonic aggregation.

| | *Size of argument set* | | | |
|---|---|---|---|---|
| | 8(18) | 16(38) | 24(58) | 32(78) |
| `short1` | 0.0032(2) | 0.0303(6) | 0.0979(10) | 0.242(14) |
| `short2` | 0.0072(3) | 0.0156(6) | 0.0238(9) | 0.0324(12) |

Our second example shows the performance of memoization and re-do for the company controls example from section 2.2.2 (note that `false < true` in the boolean partial ordering):

```
controls(X,Y) >= sum(owns(X,Y)) > 50
owns(X,Y) contains {s(X,Y,N)} :- shares(X,Y,N)
owns(X,Y) contains {s(Z,Y,N)} :- shares(Z,Y,N), controls(X,Z)=true
sum(phi) equals 0
sum({s(_,_,C)\T}) equals C+sum(T)
```

For the above program, the following figures were obtained by running the `controls` function on extensional databases of different sizes, and observing the effect of the number re-do operations on the overall execution time. The term *# of edges* in the table below refers to the size of the extensional database, i.e., the number of facts

of the relation `shares(X,Y,N)` (an in-memory database). We introduced several cyclic holding dependencies in order to exercise the re-do mechanism.

| *# of edges* | *Time* |
|---|---|
| 40 | 0.112 |
| 80 | 0.389 |
| 160 | 1.542 |
| 320 | 6.631 |

The above program requires a redo operation since `owns/2` is an argument to a monotonic function `sum/1`. Thus, when updating a memo-table entry, a set-equality check between each new approximation and the previous one is made to determine if a redo is required. As a result, the performance grows quadratically as the number of edges increases.

## *5.2. Comparisons with XSB and CORAL*

In this section we compare our implementation of SuRE with XSB and CORAL. We choose problems that are appropriate test cases for all three systems. Although SuRE is better suited to set-oriented computations, we do not consider such programs in the comparisons, because XSB and CORAL are not primarily targeted at set-oriented computations. We used XSB version 1.4.3 and CORAL version 1.2 in these comparisons, as these were the implementations available to us at the time we completed the implementation of SuRE. The reader should note that all three systems will improve in time, and hence the performance figures presented should be read with in this light. The reader should focus on program clarity, and on the rate of growth of execution times rather than on the absolute value of the execution times. However, we report actual execution times in order to provide some sense of their relative performance.

*Comparisons with XSB*

The XSB version of the shortest distance problem using `bagMin` is shown below. The program is comparable in conciseness to the SuRE formulation with partial-order clauses. XSB supports aggregation using the HiLog predicates `bagReduce/4` and `bagPO/3`. The predicate `bagMin/2` is defined using `bagReduce/4` and computes the minimum value of a bag of results. By implementing `bagReduce/4` with tabling it provides efficient access to the elements of a bag. Note that a bag is not explicitly constructed; only the running minimum, maximum, sum, etc., is maintained in the memo-table.

```
:- import bagMin/2 from aggregs.
:- hilog short.

short(X, Y)(D) :- edge(X,Y,D).
short(X, Y)(D) :-
    bagMin(short(X,Z), D1),
    edge(Z,Y,D2),
    D is D1 + D2.

short2(X,Y,D) :- bagMin(short(X,Y),D).
```

The behavior of the XSB `bagMin/2` operation for recursive programs is of special interest. As D.S. Warren describes in [37], for recursive programs operating on cyclic data, such as this one, the `bagMin/2` operation must return an answer before it has seen all elements. Thanks to the monotonicity of the `min` operator, a well-defined answer results without having to enumerate the infinitely-many paths. However, the performance of this program depends on the scheduling strategy [37], i.e., which solution path is explored first. Compared with the SuRE implementation, XSB runs 3-6 times faster although the growth rate of the two execution times are quite comparable.

| | *# of edges* | | | | |
|---|---|---|---|---|---|
| | 138 | 218 | 298 | 418 | 538 |
| SuRE/`short2` | 0.0569 | 0.097 | 0.136 | 0.185 | 0.296 |
| XSB/`short2` | 0.020 | 0.029 | 0.039 | 0.044 | 0.050 |

Following is the XSB version of company-controls problem and its performance figures. In this example, we expect that the XSB program will perform even better than the corresponding SuRE program because the latter will construct a set. Unlike the shortest-distance example, whose performance under XSB is dependent upon the scheduling strategy, the performance of the above program is not affected by the scheduling strategy because the aggregate operation here is `sum`.

```
:- import bagReduce/4 from aggregs.
:- hilog  sum.
:- table(owns1/3).

sum(X,Y,Z) :- Z is X+Y.
bagSum(Call,Var) :- bagReduce(Call, Var, sum, 0).

controls(X, Y, A) :-
   bagSum(owns(X,Y), Total),
   (Total > 50 -> A = true; A = false).

owns(X, Y)(N) :- owns1(X, Y, N).

owns1(X, Y, N) :- shares(X, Y, N).
owns1(X, Y, N) :- controls(X, Z, true), shares(Z,Y,N).
```

In the SuRE version of company controls problem, we express `controls` as a partial order clause and `owns/2` as a `contains` clause. Both aggregation and monotonicity are built into the control of SuRE. However, as the table below shows, it doesn't perform as well since `owns/2` still constructs a set.

| | *# of edges* | | | | |
|---|---|---|---|---|---|
| | 40 | 80 | 160 | 240 | 320 |
| SuRE/`controls` | 0.112 | 0.389 | 1.542 | 3.40 | 6.631 |
| XSB/`controls` | 0.020 | 0.025 | 0.040 | 0.051 | 0.081 |

Finally, note that, unlike the `perms` function of section 5.1.1, the performance of the SuRE function `controls` cannot be improved through the use of bags. The reason is that, when working with bags, monotonic updating of memo-tables may sometimes not reach a finite fixed-point for problems involving circular constraints

(such as those that arise in the company-controls problem) even though a finite fixed-point can be obtained when working with sets. For example, while the addition of an element 1 to a set {1,2,3} results in the set {1,2,3}, in the case of bags, this will result in a new bag {1,1,2,3}. Thus, while a finite fixed-point {1,2,3} might be reached in the former case (sets), this is precluded in the latter (bags): since the bag {1,1,2,3} is different from the previous one, {1,2,3}, a re-do will be initiated, causing increasingly large bags of the form {1,1,1,2,3}, etc., to be computed.

*Comparisons with CORAL*

First we show the use of aggregation in CORAL, which allows rule-level control through the *aggregate_selection* annotation. For the shortest-distance example, we use the following program, which is taken from CORAL release 1.2, and modified only slightly to conform to the SuRE version of the same problem.

```
module declad_eg6a.
export short2(bbf).

cost(X,Y,C) :- edge(X,Y,C).
cost(X,Y,C) :- edge(X,Z,C1), cost(Z,Y,C2), C=C1+C2.

@aggregate_selection cost[bbf] (X,Y,C) (X,Y) min(C).

short2(X,Y,min(<C>)) :- cost(X,Y,C).

end_module.
```

By specifying `cost(X,Y,C)` to be aggregate_selection the system checks for `cost` facts with the same `X,Y` combination and deletes all facts whose `C` value is not the smallest. This annotation is important not only for efficiency but also termination: without it the program may run forever, generating cyclic paths of increasing length [29]. That is, at each iteration it filters out unnecessary tuples, and thus minimizes the cost of the next iteration. As a result, the grouping in `short2` operates only on the path between `X` and `Y` whose cost is the least. Reference [28] states that the aggregate_selection is an operational modification of duplicates-checking, and, in general, there is no declarative semantics for programs using these features. In contrast, SuRE uses the partial-order clause to formulate the same problem, and programs have a well-defined semantics as long as monotonic functions in the appropriate partial orders are used.

Another important control annotation of CORAL is the specification of *ordered search*. This is an evaluation mechanism that orders the use of generated subgoals in a program. It can be used to evaluate a large class of programs with negation by maintaining information about dependencies between subgoals, and to avoid a lot of redundant computations by hiding subgoals when only a single answer to a query is needed [28].

The following table compares the SuRE and CORAL versions of the short-distance program. The figures for CORAL below is with aggregate_selection and making use of separate modules for program and data. The program uses bottom-up evaluation and ordered search, along with grouping and recursion. The growth

rates of both versions are comparable, although the current version of the SuRE system tends to take less time.

| | *# of edges* | | | | |
|---|---|---|---|---|---|
| | 138 | 218 | 298 | 418 | 538 |
| SuRE/`short2` | 0.0569 | 0.097 | 0.136 | 0.185 | 0.296 |
| CORAL/`short2` | 0.21 | 0.4 | 0.53 | 0.96 | 1.70 |

The following data is obtained from running the shortest-distance program on a randomly generated tree and dag with 120 nodes and 495 edges. This data set is taken from the CORAL release (files edge.tree.F and edge.dag.F respectively). When run in *default mode*, CORAL takes more execution time for the DAG. This may be explained by the fact that bottom-up evaluation requires more iterations for certain data structures, especially when there are more elements in the transitive closure.

| | SuRE | CORAL | |
|---|---|---|---|
| | | *aggregate_selection* | *default* |
| *tree* | 0.0349 | 0.13(1.53) | 0.14(1.53) |
| *dag* | 0.125 | 0.41(7.94) | 4.14(12.29) |

The figures in parentheses are for the single-module case and show the performance penalties that might occur without modularization. Since the tree and dag contain no cycle, SuRE will not perform redo's although it will check if one is needed each time the memo-table is updated.

While aggregate_selection improves efficiency significantly as the above test shows, its use is somewhat limited to cases where we wish to retain just one of several computed facts. In the case of company-controls example shown below [29], aggregate_selection cannot be used because the aggregate operation is 'sum': we need to sum up the shares from all the intermediate facts computed. As a result, an annotation *monotonic* is used instead. Since the `owns/2` relation can be cyclic, it can lead to a cycle of goals through grouping. Without this annotation, it will not terminate when generating all the shares of companies `Z` owned by `X`.

```
module company_control.
export controls(ff).

@monotonic.

controlsvia(X,X,Y,N) :- shares(X,Y,N).
controlsvia(X,Y,Z,N) :- controls(X,Y), s(Y,Z,N).

controlmin(X,Z, sum(<M>)) :- controlsvia(X,Y,Z,M).

controls(X,Y) :- controlmin(X,Y,N), N > 0.5.
end_module.
```

In the above example, no ordered search is used even in the presence of grouping with recursion because the monotonic-annotation overrides it. When tested against the SuRE version shown in the previous subsection, CORAL performs better even when a single module is used.

| | *# of edges* | | | | |
|---|---|---|---|---|---|
| | 40 | 80 | 160 | 240 | 320 |
| SuRE/`controls` | 0.112 | 0.389 | 1.542 | 3.40 | 6.631 |
| CORAL/`controls` | 0.02 | 0.04 | 0.08 | 0.11 | 0.17 |

In conclusion, while SuRE can declaratively express set-oriented operations using subset clauses and partial order clauses, it incurs a performance penalty because it must explicitly construct sets in certain cases. Not only does involve more time in set-collection, it requires a set-equality check to determine whether a re-do is needed. While one can note some similarities between SuRE and CORAL with respect to set terms, their approaches to handling aggregation is different. In CORAL an aggregate operation such as *min* is supported via aggregate_selection and a multiset operator *min*. But an aggregate operation such as *sum* which collects all the intermediate results needs to use an annotation monotonic to avoid nontermination. Whereas in SuRE the *min* operation can be expressed naturally via partial order clauses, the *sum* operation must be expressed by first forming a set (using subset clauses) and then summing its members.

XSB performs the best among the three systems on benchmark problems that are common to these systems. Much of the efficiency in CORAL comes from sophisticated hashing and indexing, since it is critical in bottom-up evaluation to perform duplicate tuple-checking and avoid unnecessary iterations. XSB also uses clever data structures (*tries*) for representing terms. We feel that SuRE can benefit from adapting some of the techniques used in XSB and CORAL, thus providing more efficient solutions while supporting set terms declaratively. Finally, we would like to stress that the performance figures for the systems shown in this section should be interpreted as snapshots at one point in their evolution. We expect that all three systems will improve in time. Nevertheless we hope that the comparisons show how typical problems can be expressed in these systems and also the relative strengths of these systems for different problems.

## 6. Summary, Conclusions and Further Work

The paradigm of subset-logic programming provides a rich collection of declarative features for processing sets. It should not be surprising that sets play a crucial role in logic programming and deductive databases, since the semantics of relational programs are, after all, given in terms of sets. By formulating a set-valued function through subset clauses, one gains the flexibility of operating on the resulting in different ways: eagerly, incrementally, as well as lazily. Subset clauses and, more generally, partial-order clauses help render clear and efficient formulations to problems requiring *setof* operations, transitive closures, and monotonic aggregation in deductive databases.

The two main implementation issues in a logic programming language are the implementation of unification and the implementation of its control strategy. In subset-logic programs, the former involves the implementation of set matching and set unification. As noted earlier, this topic has been treated in other papers, hence the main focus of this paper has been on the implementation of the control strategy. Compared with Prolog, the needed control in subset-logic programs is more complex, requiring memo-tables, re-execution of calls, as well as incremental and lazy

exploration of a search space. Notwithstanding these differences, the Warren Abstract Machine [35] has been a suitable framework for implementing these features. We summarize the salient points of our implementation:

- Apart from the instructions needed for compiling set-matching and set-unification, only a few instructions were needed for compiling the control strategy. Essentially one or two instructions were needed in order to make function calls in each of the different modes: eager, incremental, memoized (and lazy) modes.
- The main new storage structure needed is the memo-table, whose entries are monotonically updated in the appropriate partial order. On the control stack we introduce three new records: branch points (to keep track of branching in set matching and unification), and lookup points (which identify the place where a re-do is required). Finally, we make use of a redo-trail to help restore the environment for conditional subset clauses.
- Although our current implementation of subset-logic programs has not been optimized in any special way, our experiments with the implementation show that memoization and re-do are practical computational mechanisms for a set-oriented declarative language.

In this paper we have not discussed lazy evaluation of subset logic functions. As noted earlier, a basic difference between lazy evaluation in the subset-logic paradigm and the conventional functional paradigm is that the unevaluated portion of a lazy set might be some control point in the search tree of a relational program. We therefore represent this unevaluated part by a *read-only* variable, which when matched against a set constructor will result in the search tree being explored. Because this search space is explored only when this read-only variable is matched against a set constructor, the space is pruned if the read-only variable is never accessed again. Thus, by "objectifying" the search space as a set, we are able to declaratively prune the search space. A key implementation problem here is protecting the search space from being explored by normal backtracking. In our current implementation, the programmer specifies lazy exploration of a search space through an annotation, but we are examining ways of automating this decision through static analysis.

Another opportunity for static analysis is the detection of the property of distribution over union. In every non-contrived program that we have encountered, we have observed that there are two very simple tests can be used determine that a function distributes over union in some argument: (i) a set constructor, `{H\T}` or `{H\_}`, appears in this argument at the head of the clause, and the remainder set `T` is not subsequently used (e.g., as in `intersect`); or (ii) a variable appears in this argument at the head of the clause and every occurrence of this variable on the right-hand-side of the subset clause is in an argument position of a function that distributes over union in this argument position (e.g., as in `union`, trivially).

As part of our future work, we are interested in executing efficiently functions such as `short`, which are defined using partial-order clauses but whose result domain is totally ordered (e.g., integers under `<=`). When this is the case, the computational procedure can be adapted so that the search for an answer can be directed more efficiently and unproductive subexpressions can be blocked from further evaluation.

The resulting computational regime is similar to branch and bound algorithms. Such "total-order clauses" also lend themselves to operations such as relaxation and incremental recomputation. Using relaxation, we can determine, for example, the second shortest distance from a source to a destination. Incremental recomputation is useful in the database context in determining answers to queries when a small change to an extensional relation is made.

### Acknowledgments

This research was supported in part by grant CCR 9613703 from the National Science Foundation. The implementation of SuRE was carried out by Kyonghee Moon as part of her Ph.D. dissertation, and can be obtained by anonymous FTP from `ftp.cs.buffalo.edu`. The SuRE compiler was modeled after the SB-Prolog compiler developed by Saumya Debray and his colleagues. We are grateful to them for allowing us FTP access to the SB-Prolog source code.

## REFERENCES

1. H. Aït-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*, The MIT Press, 1991.
2. S. Abiteboul and S. Grumbach, A Rule-Based Language with Functions and Sets, *ACM Trans. on Database Systems*, 16(1):1–30.
3. A. Aho and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977.
4. C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur, Set Constructors in a Logic Database Language, *JLP*, 10:181–232, 1991.
5. A. Dovier, E.G. Omodeo, E. Pontelli, and G. Rossi, {log}: A Language For Programming in Logic with Finite Sets, *JLP*, 28:1–44, 1996.
6. A. Dovier and E. Pontelli, A WAM-Based Implementation of a Logic Language with Sets, *Proc. Symp. Programming Language Implementation and Logic Programming*, pp. 275-290, Springer Verlag, 1993.
7. D. Friedman, D. Wise, and M. Wand, Recursive Programming Through Table Look-up, *Proc. ACM Symp. on Symbolic and Algebraic Computation*, pp. 85–89, 1976.
8. S. Greco, D. Saccá, and C. Zaniolo, Dynamic Programming Optimization for Logic Queries with Aggregates, *Proc. Intl. Logic Programming Symposium*, pp. 575–589, MIT Press, 1993.
9. D. Jana, Semantics of Subset-Logic Languages, Ph.D. dissertation, Department of Computer Science, SUNY-Buffalo, August 1994.
10. B. Jayaraman, The SuRE Programming Framework, TR 91-011, Department of Computer Science, SUNY-Buffalo, August 1991.
11. B. Jayaraman, Implementation of Subset-Equational Programs, *JLP*, 11:299–324, 1992.
12. B. Jayaraman and D. Jana, Set Constructors, Finite Sets, and Logical Semantics, To appear in *Journal of Logic Programming*.
13. B. Jayaraman and A. Nair, Subset-Logic Programming: Application and Implementation, *Proc. Jt. Intl. Conf. and Symp. on Logic Programming*, pp. 841–858, Seattle, 1988.

14. B. Jayaraman and D.A. Plaisted, Functional Programming with Sets, *Proc. Third Intl. Conf. on Functional Programming and Computer Architecture*, pp. 194–210, Springer-Verlag, 1987.

15. B. Jayaraman and D.A. Plaisted, Programming with Equations, Subsets, and Relations, *Proc. N. Amer. Conference on Logic Programming*, pp. 1051–1068, MIT Press, 1989.

16. G.M. Kuper, Logic Programming with Sets, *JCSS*, 41(1):44–64.

17. D.B. Kemp and P.J. Stuckey, Semantics of Logic Programs with Aggregates, *Proc. Intl. Logic Programming Symposium*, pp. 387–401, MIT Press, 1991.

18. M. Liu, Relationlog: A typed Extension to Datalog with Sets and Tuples, *Proc. Intl. Symp. of Logic Programming*, pp. 83–97, MIT Press, 1995.

19. J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1987.

20. K. Moon, *Implementation of Subset Logic Languages*, Ph.D. dissertation, Department of Computer Science, SUNY-Buffalo, February 1997.

21. I.S. Mumick, H. Pirahesh, and R. Ramakrishnan, The Magic of Duplicates and Aggregates, *Proc. 16th VLDB Conference*, pp. 264–277, 1990.

22. G. Nadathur and B. Jayaraman, Towards a WAM Model for λProlog, *Proc. North American Logic Programming Conference*, pp. 1180–1198, MIT Press, 1989.

23. M. Osorio, *Semantics of Logic Programs with Sets*, Ph.D. dissertation, Department of Computer Science, SUNY-Buffalo, 1995.

24. M. Osorio and B. Jayaraman, Aggregation and Negation-as-Failure, To appear in *New Generation Computing*.

25. M. Osorio, B. Jayaraman, and D.A. Plaisted, Theory of Partial Order Programming, To appear in *Science of Computer Programming*.

26. T. Przymusinski, On the Declarative Semantics of Stratified Deductive Databases and Logic Programs, *Proc. Foundations of Deductive Databases and Logic Programming*, J. Minker (ed.), pp. 193–216, Morgan-Kaufmann, 1988.

27. I.V. Ramakrishnan, P. Rao, K.F. Sagonas, T. Swift, D.S. Warren, Efficient Tabling Mechanisms for Logic Programs, *Proc. Intl. Conf. on Logic Programming*, pp. 697-711, MIT Press, 1995.

28. R. Ramakrishnan, D. Sivastava, and S. Sudarshan, CORAL–Control, Relations and Logic, *Proc. 18th VLDB Conference*, pp. 238–250, 1992.

29. R. Ramakrishnan, P. Seshadri, D. Srivastava, and S. Sudarshan, *The CORAL User Manual: A Tutorial Introduction to CORAL*, unpublished manuscript, 1993.

30. R. Ramakrishnan, P. Seshadri, D. Srivastava, and S. Sudarshan, Implementation of the {Coral} Deductive Database System, *Proc. ACM SIGMOD Conference*, pp. 167–176, 1993.

31. K.A. Ross and Y. Sagiv, Monotonic Aggregation in Deductive Databases, *Proc. ACM 11th Principles of Database Systems*, pp. 114–126, 1992.

32. S. Sudarshan, D. Srivastava, R. Ramakrishnan, and C. Beeri, Extending the Well-Founded and Valid Semantics for Aggregation, *Proc. Intl. Logic Programming Symposium*, pp. 590–608, MIT Press, 1993.

33. T. Swift and D.S. Warren, Analysis of SLG-WAM Evaluation of Definite Clause Programs, *Proc. Intl. Logic Programming Symposium*, pp. 219–235, MIT Press, 1994.

34. A. Van Gelder, The Well-Founded Semantics of Aggregation, *Proc. 11th ACM Principles of Database Systems*, pp. 127–138, 1992.

35. D.H.D. Warren, An Abstract Instruction Set for Prolog, Technical Note 309, SRI International, 1983.

36. D.S. Warren, Memoing for Logic Progams, *CACM*, 35(3):93–111, 1992.

37. D.S. Warren, Logic Programming in XSB, pre-publication draft, August 1995.