IMPLEMENTATION OF SUBSET LOGIC PROGRAMS

Bharat Jayaraman
Kyonghee Moon

226 Bell Hall
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
U.S.A.

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

ABSTRACT

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 an earlier paper [Jay92], we discussed 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 eager enumeration of sets. Although the subset-logic paradigm differs substantially from that of Prolog, we show that few additional changes are needed to the WAM [War83] 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 who 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 was completed in 1994, 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.

KEYWORDS

subset and relational program clauses, sets, monotonic aggregation, incremental and lazy enumeration of sets, memo tables, set matching and unification, WAM extensions, run-time structures, performance analysis

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 [JP87, JN88, JP89, Jay91, Jay92, OJ93, Jan94, JJ94, JOM95]. 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. We therefore first motivate our interest in subset clauses, describe the novel implementation issues that 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 [AG91, BNST91, DOPR91, JP89, Kup90]. 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 may take 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 ***generalized 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. (Note that this definition is more liberal than the usual definition of local stratification [Prz88]). We illustrate such programs in section 2. A more formal account of the theory of subset-logic programming can be found in [Jan94, OJ93].

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*** pro-

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

gramming. This is a purely functional paradigm, and its implementation issues were mostly discussed in [JN88, Jay92]. 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 [Jay92].

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` $\in$ `S` and `T=S-{X}`, i.e., the set `S` with `X` removed [Jay92]. 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 $\cup$ 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}` $\cup$ `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` $\in$ `T`. The `{X/T}` constructor has also been advocated by several other researchers in logic programming and deductive databases [DOPR91, BNST91][2]. 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 presence of a set constructor in the head of a relational clause necessitates ***set-unification***. Our experience with the paradigm, however, shows that for many practical programs making use of sets in relational clauses, set-unification reduces to set-matching.

Subset clauses have many uses in the logic programming context: They serve as a declarative alternative to ***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 (by a membership goal) or collectively (by an equational goal). In the latter case, one further has the flexibility of working lazily or eagerly. 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. We show that lazy enumeration of solutions is the key to pruning the search space in generate-and-test problems, while eager enumeration corresponds to the `setof` construct. We showed in our earlier work [JP87, Jay92] that when a function ***distributes over union*** with respect to 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 is currently given as an annotation in our implementation, but it can be inferred automatically in most practical cases.

Subset clauses are also useful in the deductive database context: They help render clear

[2]In [Jay92] 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.

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 [MPR90, KS91, RS92, Van92, GSZ93, SSRB93]. 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 [Van92]. Moreover, for many problems requiring aggregate operations to be performed, the concept of monotonic functions is necessary [RS92]. 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.

1.2 Implementation Issues

The Warren Abstract Machine (WAM) [Ait91, War83], designed by D.H.D. Warren, has proven to be a robust framework for implementing Prolog as well as several variants of the language [NJ89, DP93, Jay92, SW94]. 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 [SW94] for the class of definite clause (relational) programs. This improvement is attributed to the efficient data-structure representation 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 [Jay92] and set-unification in [DP93]—and therefore we discuss them only briefly here. Both operations can in general result in multiple maximally general matches/unifiers, and hence may cause additional branching in the search process. We showed in [Jay92] 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, we introdued 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, corresponding to *eager* evaluation, *incremental* evaluation, *lazy* evaluation, or *memoized* evaluation. We briefly discuss each of them below.

When a function is invoked eagerly 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 not ever form the entire set, 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 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. We solve this problem by designating the choice point denoted by a read-only variable as a *lazy choice point*, and precluding it from being the target of normal backtracking.

Memoization in subset-logic programs differs from the way it is used in both functional and logic programming languages: In functional programming, memoization is traditionally used to detect dynamic common subexpressions [FWW76], and serves as a way to obtain an efficient implementation of dynamic programming algorithms. In addition to the above motivation, memoization in logic programs is used in order to detect *circular* loops that arise in programs for problems such as transitive closure [SW94, War92]. 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. 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 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 [SW94]. 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. We provide performance figures for a variety of programs to illustrate various facets of our implementation of memoization. We show that the overhead of memoization in subset-logic programs is minimal, a result that is similar to the one reported in [SW94], 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. Finally 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).

The rest of this paper is organized as follows: section 2 describes the syntax and presents 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; and section 4 presents the new instructions of the abstract machine, examples of compiled code, and performance figures from our current implementation; finally, section 5 presents conclusions and areas of further work. We assume some familiarity with the implementation of logic programs, especially the Warren Abstract Machine [Ait91,War93].

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 [Jay92]. 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, monotonic aggregation, and the remaining forms of set enumeration here; We also introduce partial-order clauses in this subsection.

2.1 Unconditional Subset Clauses

Unconditional subset clauses are of the form

f(terms) `contains` *expr*

where each variable in *expr* also occurs in *terms*. The syntax of *term* is:

$term$ `::=` $variable \mid constant \mid constructor(terms) \mid set$

set `::=` `phi` $\mid variable \mid \{term \backslash set\}$

$terms$ `::=` $term \mid term$ `,` $terms$

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:

$expr$ `::=` $variable \mid constant \mid$ `phi` $\mid \{expr/expr\} \mid constructor(exprs) \mid function(exprs)$

$exprs$ `::=` $expr \mid expr$ `,` $exprs$

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 [JP89, Jan94]. 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*.

2.1.1 Iteration via Set Matching

As noted in section 1, the term `{X\T}` matches a set `S` such that `X` $\in$ `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 this set constructor, matching `{X\T}` against a ground set `{a,b,c}` yields three different substitutions:

$$\begin{array}{ll} \texttt{X} \leftarrow \texttt{a}, & \texttt{T} \leftarrow \{\texttt{b},\texttt{c}\} \\ \texttt{X} \leftarrow \texttt{b}, & \texttt{T} \leftarrow \{\texttt{a},\texttt{c}\} \\ \texttt{X} \leftarrow \texttt{c}, & \texttt{T} \leftarrow \{\texttt{a},\texttt{b}\} \end{array}$$

There are only finitely many matches of a set-term `{X\T}` against any set constructed from `phi` and `{_\_}`. 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 `union(X1,X2)` $=$ `X1` $\cup$ `X2`. Aside: the term `{X/T}` $=$ `union({X},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 ground goal `intersect({1,2,3},{2,3,4})`, we obtain `intersect({1,2,3},{2,3,4}) contains {2}` and `intersect({1,2,3},{2,3,4}) contains{3}`. By the collect-all assumption, we infer that `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 either argument is `phi`.

The use of remainder sets in set-matching is illustrated by the following function definition, which takes as input a set of *propositional clauses* (i.e., a set of set of literals) and returns 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 `{A,B\_}` is an abbreviation for `{A\{B\_}}`.

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 [Jay92].

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. Set-valued functions that compute some aggregate property of a set, e.g., `resolvents` and `perms`, do not distribute over union. Functions such as `union`, `setproduct`, and `intersect`, which are defined in terms of the elements of their argument sets do distribute over union in both of their arguments. (Note that `distr` distributes over union in its second argument). In recognition of this property, we distinguish two ***modes*** of calling 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, e.g.,

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

When a function ***distributes over union*** with respect to one of its arguments, we can avoid forming the intermediate set in this argument position and thereby avoid the check 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 [Jay92].

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 of memoization commonly arises in recursive subset clauses of the form

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

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 $n-2$. Another example is the following function that computes the different partitions of a set:

```
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})
```

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 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 [AU77]:

```
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` is monotonic in its first argument, and hence its use in the body of `out` is legal. 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 [AU77], wherein an imperative program is written to compute the desired sets using a bottom-up strategy.

We require subset clauses to obey a ***generalized 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 more general than the one in [Prz88] because we permit disciplined forms of circularity to occur in the dependency graph of function calls.

2.2 Conditional Subset Clauses

Conditional subset clauses have the form

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

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

condition `::=` *goal* | *goal*, *condition*

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

The declarative reading of a conditional subset clause is similar to that for unconditional subset 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 to obey the local stratification condition [Prz88], and, as before, all function calls must obey the generalized local stratification condition stated at the end of section 2.1.2. We treat negated goals by negation as failure [Llo87]. 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 must be invoked with ground terms as arguments—because negation-as-failure may in general be unsound for nonground goals.

We illustrate these features by giving three 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; (ii) monotonic aggregation; and (iii) lazy exploration of a search space. In discussing (i), we introduce general relational clauses and the use of set terms in them; in discussing (ii), we introduce partial-order clauses, a straightforward generalization of subset clauses; and in discussing (iii), we explain the use of set enumeration goals.

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 $\{\_\backslash\_\}$, 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, by a goal such as `member(b, {a, b, c})`; or to generate the elements of a set one at a time, by a goal such as `member(X, {a, b, c})`; or to insert an element in a set, by a goal such as `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 $\{\_\backslash\_\}$ constructor were used in the above clause definition, 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`. On the other hand, there are infinitely many solutions to the goal `list_to_set({1}, L)`, namely, `L` $\leftarrow$ `[1]`, `L` $\leftarrow$ `[1,1]`, `L` $\leftarrow$ `[1,1,1]`, etc. In general, it is preferable to use the $\{\_\backslash\_\}$ 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 [JP89]. 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 by the following subset clause and goal.

```
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

$$\begin{aligned} & \{\texttt{pair}([\ ],[1,2,3])\} \cup \{\texttt{pair}([1],[2,3])\} \cup \{\texttt{pair}([1,2],[3])\} \cup \{\texttt{pair}([1,2,3],[\ ])\} \\ = & \{\texttt{pair}([\ ],[1,2,3]), \texttt{pair}([1],[2,3]), \texttt{pair}([1,2],[3]), \texttt{pair}([1,2,3],[\ ])\} \end{aligned}$$

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 us 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 [RS92], 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 {N} :- shares(X,Y,N).
owns(X,Y) contains {N} :- shares(Z,Y,N), controls(X, Z) = true.
```

This example also illustrates the use of an aggregate operation, `sum`. The function `controls(X,Y)` returns `true` if company `X` controls `Y`, and `false` otherwise. 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%.[3] 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 are satisfied for a well-defined semantics.

Our last example shows another example of a partial-order clause. 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 perhaps most 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. 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 yet support annotations that specify total-ordering.

2.2.3 Lazy Enumeration

There are three forms of set enumeration in the subset-logic paradigm:

1. Incremental enumeration of the elements of a set is specified by membership goal of the form *term* `in` *f*(*terms*), where *terms* must be ground. Here, the elements of the set denoted by *f*(*terms*) are generated one at a time, and each resulting term is matched against *term*. The use of this goal is straightforward, and hence we do not discuss it further.

2. Eager enumeration of a set is specified by an equality goal of the form *f*(*terms*) `=` *term*, where the function call must have ground terms. Here, the set denoted by *f*(*terms*) is generated all at once, and it is matched against *term* (which must be a set). The use of this goal is also straightforward. All top-level functional expressions presented thus far are examples of eager set enumeration. For example, the goal `|?- partitions([1,2,3])` is actually an abbreviation for `|?- partitions([1,2,3]) = X`, for any variable `X`.

[3] It would be more appropriate in this example to build a multiset (instead of a set) as the argument to `sum`. However, one can re-program this problem retaining the use of sets, by tagging each percentage value with the corresponding company-name.

3. Lazy enumeration of the elements of a set is specified by an annotated equality goal of the form `lazy` *f*(*terms*) `=` *term*, where the function call must have ground terms. (Ideally, the annotation should be inferred by a compiler, but this cannot be done in all cases, as the problem is similar to strictness analysis). Here, the set denoted by *f*(*terms*) is generated lazily, and it is matched against *term* (which must be a set).

We illustrate lazy enumeration with a simple example—the N Queens problem. (While there are more efficient algorithms for solving this problem, this example is used only for illustrating the concept). The predicate `solve` below employs what appears to be an extremely naive strategy of first generating all possible N×N board configurations and then testing for safe board configurations. However, due to lazy evaluation, the set of all board configurations is represented as a *lazy set*, and many of them will not be examined. The resulting program preserves the modularity of a generate-and-test algorithm, but yields the efficiency of a more conventional algorithm of placing queens column-by-column and backtracking only when a conflict occurs.

```
solve(N,Ans) :-  generate(N,N,Boards),
                 test(Boards, phi, Ans).

generate(1,N,[R])   :- lazy rows(1,N) = R.   % lazy generation of row positions
generate(I,N,[R|B]) :- I >1,
                    lazy rows(I,N) = R,     % lazy generation of row positions
                    I-1 = J,
                    generate(J,N,B).

rows(I,N) contains {pos(I,J)} :- digit(1,N,J).

digit(K,N,K).
digit(K,N,J) :- K+1 = L,  L =< N,  digit(L,N,J).

test([],            Safe, []).
test([{Queen/_}|B], Safe, [Queen|Rest]) :-
                not attacking(Queen, Safe),
                test(B, {Queen/Safe}, Rest).

attacking(pos(C1,R1), {pos(C2,R2)/_}) :- overlaps(C1,R1,C2,R2) = true.

overlaps(C1,R1,C2,R2) = eq(R1,R2) or eq(abs(C1-C2),abs(R1-R2)).
                                  % this is an example of an equational clause
```

We now explain briefly what we mean by a lazy set. The goal `lazy rows(1,4) = R`, for example, results in the lazy set

$\{\mathsf{pos}(1,1)/*\}$

in which only one element has been computed. The $*$ represents the unevaluated remaining portion of the set; it is essentially a pointer to a *choice point* (the second clause of `digit`), which is used *when needed* to produce subsequent elements. Thus the goal `generate(4,4,Boards)`, for example, will result in the following solution for `Boards`:

$$\texttt{Boards} \;=\; [\{\texttt{pos}(4,1)/*\}, \{\texttt{pos}(3,1)/*\}, \{\texttt{pos}(2,1)/*\}, \{\texttt{pos}(1,1)/*\}]$$

The *i-th* lazy set of the above list represents the row positions of the *i-th* column. We briefly explain how a lazy set is explored. Suppose the above list were passed by `solve` as the first argument of `test`. The second clause-head of `test` would successfully unify (the variable $\texttt{Queen}^1$ would be bound to `pos(4,1)`), the goal `not attacking(pos(4,1),phi)` would then succeed, and the recursive call $\texttt{test}([\{\texttt{pos}(3,1)/*\}, \{\texttt{pos}(2,1)/*\}, \{\texttt{pos}(1,1)/*\}], \{\texttt{pos}(4,1)\}, \texttt{Rest})$ would then be made. The second clause-head of `test` would again successfully unify (the variable $\texttt{Queen}^2$ would be bound to `pos(3,1)`), but now the goal `not attacking(pos(3,1),{pos(4,1)})` fails, and we backtrack to obtain the next match for variable $\texttt{Queen}^2$. It is at this point that the next element of the lazy set $\{\texttt{pos}(3,1)/*\}$ gets generated. In this manner the search proceeds and the lazy sets are gradually constructed. Note that, since the lazy sets $\{\texttt{pos}(2,1)/*\}$ and $\{\texttt{pos}(1,1)/*\}$ are not explored for the partial board configuration with queens in `pos(4,1)` and `pos(3,1)`, we have effectively pruned this search space.

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 [Jay92], 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 extension to handle the conditional subset clauses, especially monotonic aggregation and lazy enumeration.

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.

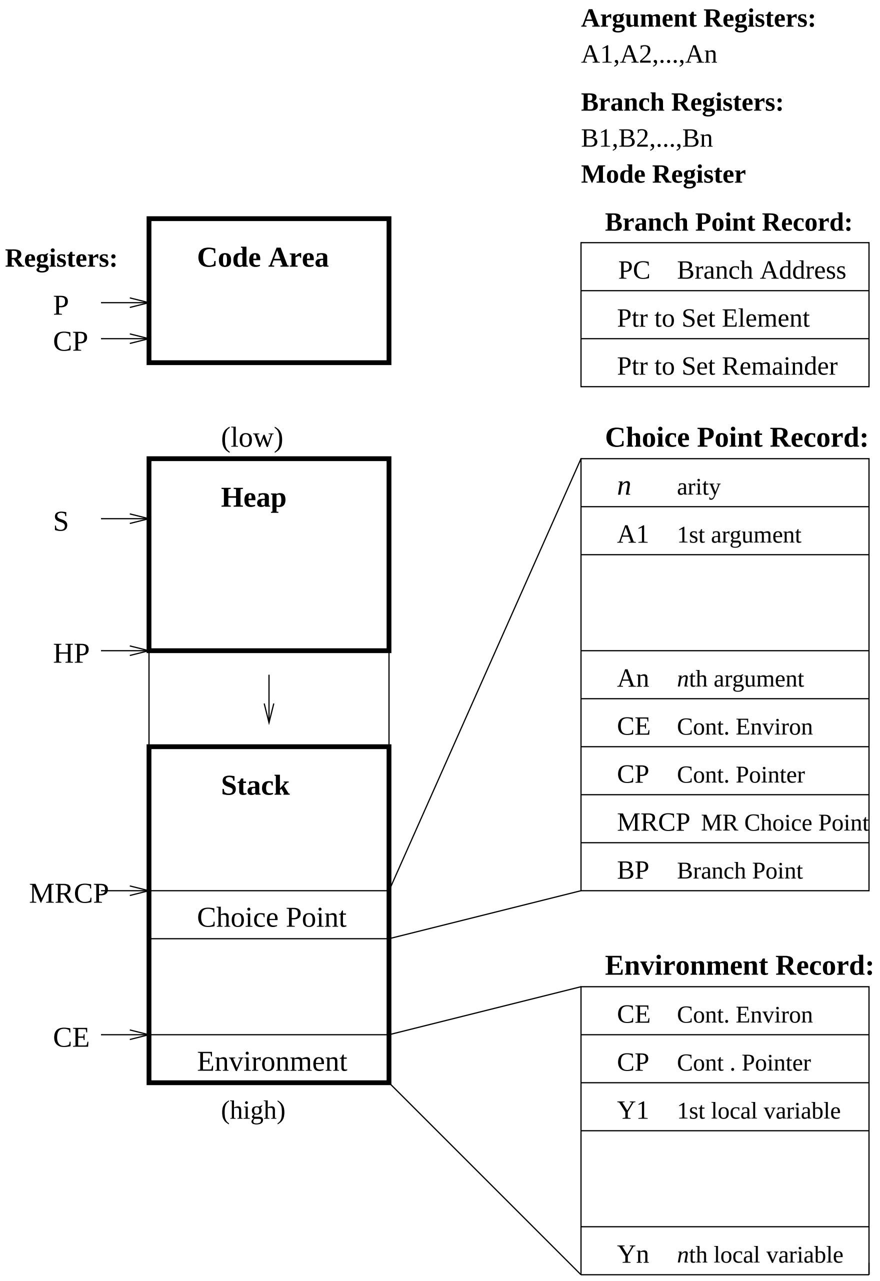

Figure 3.1: Basic Abstract Machine

- *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, as long as there is at least one set constructor at the head of this clause. In order to keep track of the state of match of set terms of the form `{H1\T1}`, we augment the abstract machine with a set of *branch registers*, where each branch register records the address of the instruction (see section 4) for a distinct occurrence of the set constructor. We also push on the control stack a new kind of record, called a ***branch point*** record, which saves the content of the appropriate branch register as well as the current bindings of `H1` and `T1`, since new matches for `H1` and `T1` are constructed from previous matches (see [Jay92] 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***.

- *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. Since a call-one invocation returns one element of a set at the time, 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, 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 abstract computational model by means of a 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 ***flattten*** 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 more 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 ← {10} ∪ S1	ϕ
h(1)=S1	S1 ← {20} ∪ S2	{g(1)=phi}
(L) g(1)=T1, p(T1)=S2	T1 ← phi	{g(1)=phi, h(1)=phi}
p(phi)=S2	S2 ← phi	{g(1)=phi, h(1)=phi}
(R) g(1)=T1, p(T1)=S2	T1 ← {10,20}	{g(1)={10,20}, h(1)={20}}
p({10,20})=S2	S2 ← {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 reach(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.

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` 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.

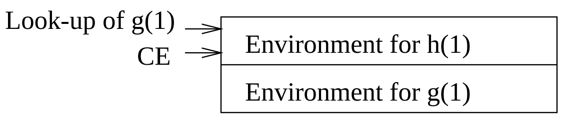

Figure 3.2a

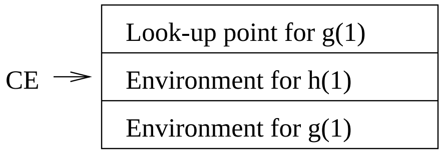

Figure 3.2b

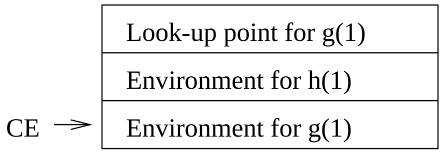

Figure 3.2c

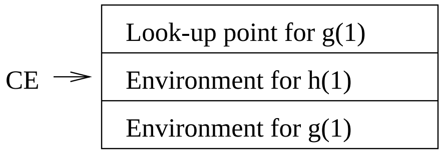

Figure 3.2d

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 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 triggered the re-do are discarded; they will be re-created if necessary 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 update 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).

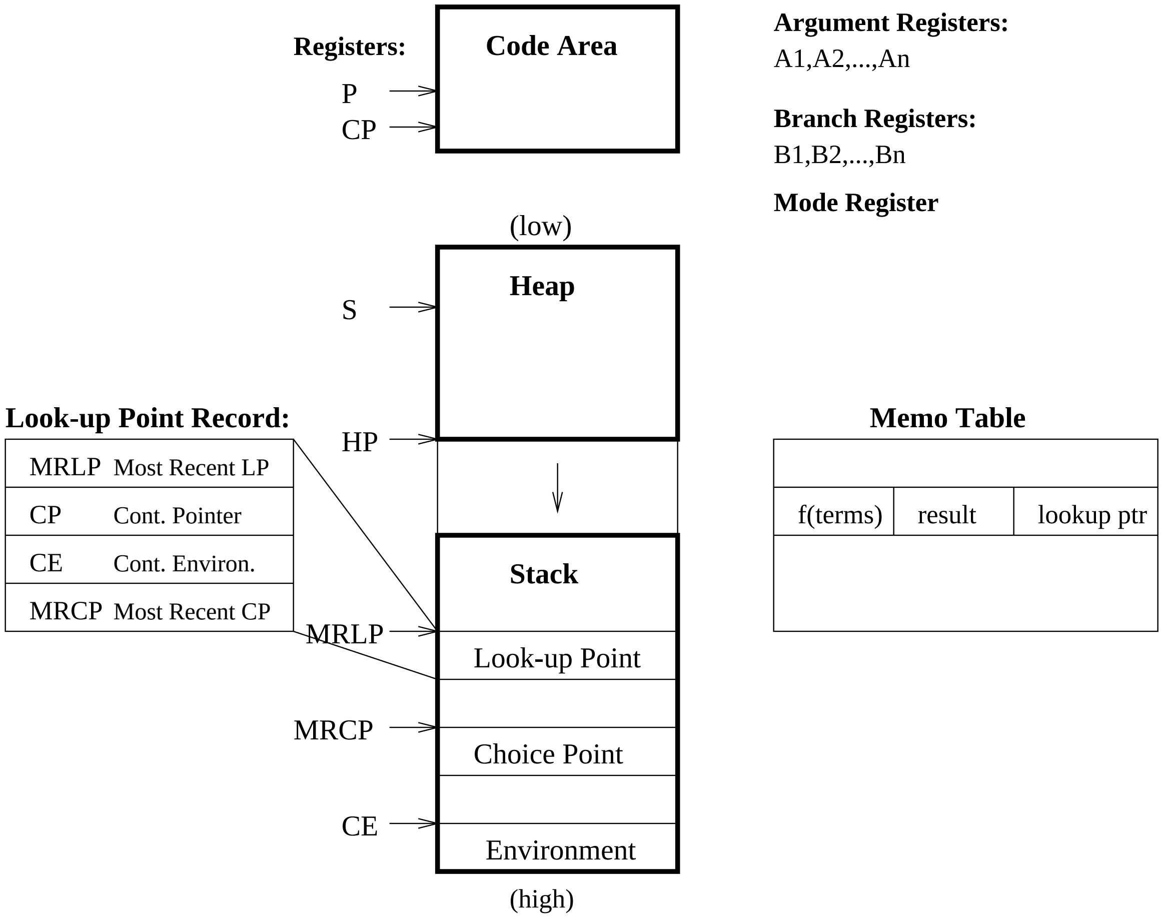

Figure 3.3: Abstract Machine for Memoization and Re-Do

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 two new features: relational goals and set enumeration 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. We discuss this topic in section 3.3.1. With respect to set enumeration goals, the main feature of interest is lazy enumeration, whose implementation we discuss in section 3.3.2.

3.3.1 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).
```

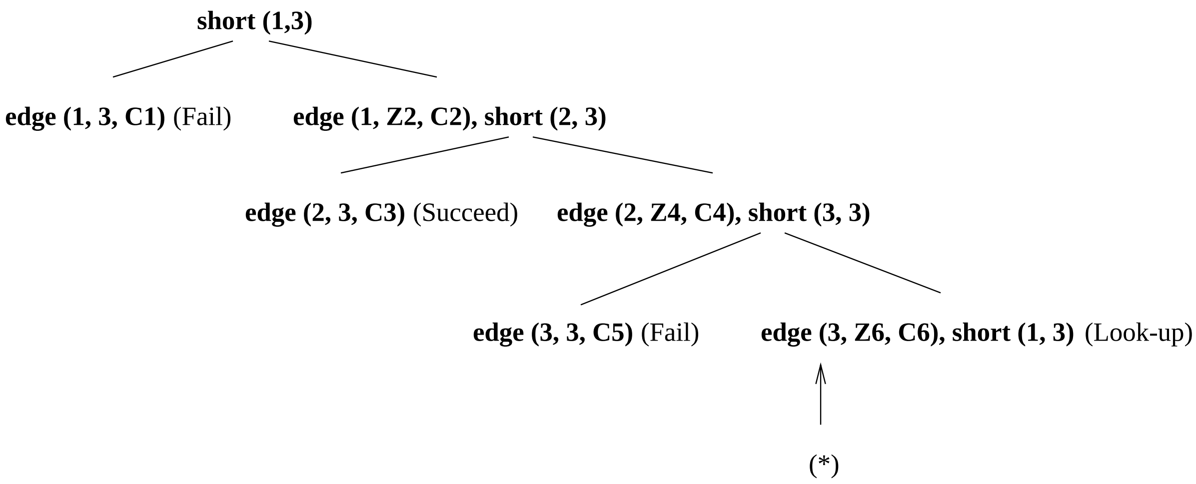

Figure 3.4

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 fails. 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*. 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.

3.3.2 Lazy Enumeration

The implementation of a lazy set involves the use of a *read only* variable to denote the unevaluated portion of the lazy set, i.e., corresponding to '*'. Whenever this read-only variable is matched against a set constructor, more elements of the lazy set are to be generated. This technique is similar to that employed in concurrent logic programming languages such as Parlog [CG86]; the main difference is that in concurrent languages these read-only variables are introduced by the programmer and their bindings are explicitly given by the program. In contrast, the read-only variable in our context must be introduced by the implementation and it refers to some resumption point from where execution must proceed in order to obtain the next element of the set. We illustrate with a simple example.

```
nums(N) contains {N}.
nums(N) contains S  :- N+1 = K,  lazy nums(K) = S.

test({X\_}) contains {X} :-  X = 2.

|?-  lazy nums(1) = S,  Answer in test(S).
```

In the above example, the set of numbers from `nums` is generated lazily. We briefy explain how the first two numbers are generated. Initially, the top-level lazy call `nums(1)` results in the variable `S` bound to the pair `{1/R1}`, where `R1` is a read-only variable. When this call completes, the implementation records an association between `R1` and the most recent choice point created (which points to the second clause of `nums`), as this is the point from where further elements of the set should be computed. Next, the call to `test` is made, which results in the variable `X` being bound to `1`. The test `1 = 2` fails, and control transfers to the most recent choice point, which is the branch point associated with `{X\_}` in `test`. Matching `{X\_}` with the read-only variable `R1` now causes the generation of the next integer from `nums`, so that `R1` gets bound to the pair `{2/R2}`, where `R2` is a new read-only variable. We will refer to the backtracking that is triggered when matching with a read-only variable as *lazy-backtracking*. As before, the implementation records an association beween `R2` and the resumption point within `nums`. The variable `X` in `test` is now bound to 2; the test `2 = 2` succeeds, and the variable `Answer` is bound to `2`.

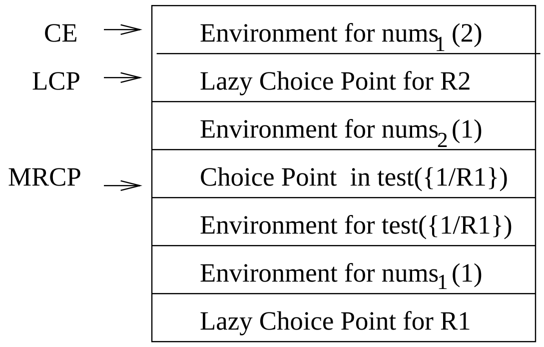

Figure 3.5

A key concern in the implementation of lazy enumeration is to prevent normal backtracking from exploring the search space associated with a read-only variable. In order to protect this search space from being traversed by normal backtracking, the choice point associated with a read-only variable is designated as a *lazy choice point*, and it cannot be the target of normal backtracking. This designation occurs when the read-only variable is set to point to this choice point, i.e., when the lazy call completes. When this designation is made, the MRCP is also updated to point to the most recent choice point *below* the environment of the lazy call. We introduce a new global register LCP to point to the current *lazy choice point* under consideration. In this way, the lazy search space is protected, and any new environment or choice point is created above the LCP. Note that there is no need to chain together lazy choice points, because the only way to access them is through the read-only variables. Figure 3.5 shows the execution state of the above program after the top-level goal has succeeded. Note that the lazy choice point for R2 protects the environment for $\texttt{nums}_2\texttt{(1)}$, i.e., the environment created for the second clause of nums.

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

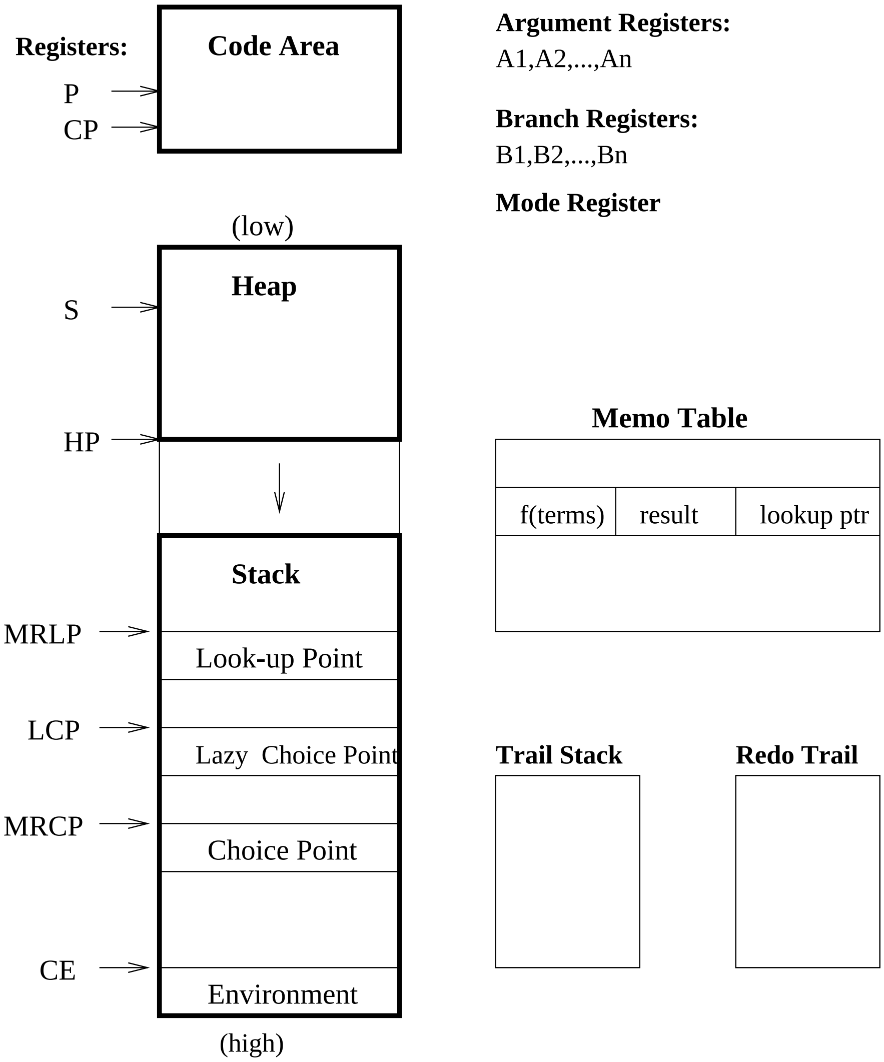

Figure 3.6: Abstract Machine for Subset-Logic Programs

4. IMPLEMENTATION AND PERFORMANCE ANALYSIS

We first describe in section 4.1 the instructions of the abstract machine for subset-logic programs and illustrate their use in compiling sample programs. In section 4.2 we report on the run-time performance of typical subset-logic programs, and discuss the efficiency of memoization and re-do.

4.1 Instruction Set and Compiled Code

In addition to the instructions of the WAM, we need instructions for compiling the set constructors, the different kinds of calls for enumerating sets, memo-table insertion/lookup, and re-do. The new instructions fall into five main classes: *get*, *adj*, *match*, *procedural*, and *choice* instructions. These classes of instructions have been treated in our earlier paper [Jay92], which dealt with only the basic abstract machine of section 3.1, i.e., for unconditional subset clauses without memoization and re-do. The new instructions for the full paradigm of subset-logic programs primarily lie in the *procedural* class. We briefly explain all these classes of instructions in order that the compiled codes can be better understood. Basically, terms appearing at the head of a subset clause will be compiled using *match* (rather than *unify*) instructions, while terms appearing at the head of a relational clause will be compiled as in Prolog. All equational goals and set enumeration goals will be compiled assuming that their arguments are ground, while relational goals will be compiled as in Prolog.

- The *get* instructions are used to compile the argument occurring in the head. Since every argument to a subset clause is ground, we can identify the *read* and *write* modes of WAM's *get* instructions at compile-time. In short, unification instructions in the read mode will be compiled into *match* instructions and those in write mode—appearing on the right-hand side of a clause—will be compiled into *store* instructions[4], to build a term. Except for a set term, every structured term occurring in the head of subset clauses will be compiled into a *get* instruction followed by *match* instructions.

- In order to compile the set constructors appearing at the head of a subset clause, we introduce three instructions—`adj_set_head`, `adj_set` and `adj_set_with_copy`—each of which treats a different case in which the remainder-set is used. 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 [Jay92]. 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. 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. Finally, we note that whenever these instructions match a read-only variable, they initiate lazy-backtracking.

[4] This instruction set is similar to that given in [Ait91].

- The procedural instructions of the WAM are augmented with four set enumeration instructions: `call_one`, `call_all`, `call_lazy`, and `execute_lazy`. We also introduce two instructions for memoized calls—`call_memo` and `execute_memo`—both of which create a memo-table entry if it is the function is being called for the first time on these arguments or perform a lookup if it is not the first time. The compiled code for each subset clause ends with a `collect` instruction whose behavior depends upon the calling mode (given in the mode register): In the call-all mode, it is responsible for constructing the resulting set and removing duplicates; in the call-one, it simply exits back to the caller; in the call-lazy mode, it is responsible for ensuring that a read-only variable has been placed at the tail of the resulting set; and, in the call-memo mode, it updates the memo-table and may initiate re-do. The collect instruction is also responsible for eventual deletion of the environment record.

- As in WAM, a program clause is indexed on its first argument. In addition to Prolog choice instructions, we have the `try_sub_and` instruction, which links together two or more subset clauses can potentially match a given function call, and it creates a choice-point frame. A `save_choice_point` instruction is placed immediately after the compiled instructions for the head of each subset clause.

We illustrate the compiled code for two typical subset-logic programs. Our first example illustrates the code for a transitive closure problem:

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

Note that the second clause of the `reach/1` definition is flattened as follows:

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

The compiled code is as follows, and should be self-explanatory in view of the comments alongside each instruction. The `execute_memo` instruction performs a memoized call.

```
reach/1: label            reach/1
         try_sub_and      L1
         allocate         3
         getvariable      Y1, A1         % reach(V)
         save_choice_point
         storeindvar      Y2, A2         % contains
         putset           Y3             % {
         storevalue       Y1             % V
         storephi                        % }
         collect          Y2, Y3         % ∪ V
L1:      label            reach/1
         allocate         4
         getvariable      Y1, A1         % reach(V)
         save_choice_point               % contains
         storeindvar      Y2, A2         % S1 :-
         putvalue         Y1, A1         % edge(V,
         putvariable      Y3, A2         % W),
```

```
        call            edge/2
        putvalue        Y3, A1        % reach(W)
        putvariable     Y4, A2        % = S2
        execute_memo    reach/1
        collect         Y2, Y4        % S1 := S1 ∪ S2
```

Our second example illustrates the code for a program requiring lazy enumeration:

```
nums(N) contains {N}
nums(N) contains S :- N+1 = K, lazy nums(K) = S
```

The annotated compiled code below should again be self-explanatory, the main new feature being the use of the `execute_lazy` instruction. Note that the `collect` instruction is responsible for ensuring that the returned set has a read-only variable in its tail when the call is made in the call-lazy mode.

```
nums/1: label           nums/1
        trysuband       L
        allocate        3
        getvariable     Y1, A1        % nums(N)
        savechoicepoint               % contains
        storeindvar     Y2, A2
        putset          Y3            % {
        storevalue      Y1            % N
        storephi                      % }
        collect         Y2, Y3
L:      label           L
        allocate        4
        getvariable     Y1, A1        % nums(N)
        savechoicepoint               % contains :-
        storeindvar     Y2, A2        % S
        putvalue        Y1, A1        % N
        putconst        int, 1, A2    % + 1
        putvariable     Y3, A3        % = K
        callsys         add
        putvalue        Y3, A1        % nums(K)
        putvariable     Y4, A2        % = S1
        executelazy     nums/1
        collect         Y2, Y4        S := S ∪ S1
```

4.2 Performance Figures and Analysis

We now discuss the results of an implementation of the foregoing ideas and techniques. All of the examples shown in this paper were tested out using this implementation—we have taken very minor liberties in changing the SuRE syntax for greater readability in this paper. The SuRE compiler[5] was written in Quintus Prolog and it generates "SLAM code," examples of which were given in the previous subsection. This code is emulated by a run-time system which was written in C.

One of the foremost implementation issues is the representation of sets. It is well-known that sets can be represented in many ways—ordered lists (as in Prolog), balanced trees, hash-tables, etc.—and that no representation is ideal for all applications. We use a list representation for sets because it very conveniently supports the two most common operations on sets in this paradigm: iteration over the elements of a set, and formation of remainders. Our representation is based on the premise that we can often defer the checks for duplicates, and hence an unsorted list representation suffices. This is borne out by the fact that many functions operate element-at-a-time on an input set (i.e., distribute over union in this argument set). In our earlier paper [Jay92], we compared the call-one and the call-all modes of invoking functions, and showed that the former generally leads to much better performance.

In this paper, we concentrate on the performance of memoization and re-do, which are the main new features of the implementation. Since we organize the memo-table 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 $\alpha < 1$. However, 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—note that 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 given in this paper. 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 Sparc 10 Model 51 (with 64 MB RAM). Each timing shown for a particular input and program is actually the average of several runs of the program on this input on a lightly loaded machine.

4.2.1 Performance of Memoization

First we consider an example where memoization is used purely for efficiency reasons. The function `parts` below is reproduced from section 2.1.2.

```
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. Not only does it save running time, it also helps avoid having to construct

[5] The SuRE compiler was modeled after the SB-Prolog compiler developed by Saumya Debray and his colleagues, and we are grateful to them for allowing us FTP access to the SB-Prolog source code.

large intermediate sets, thereby using less heap space as well. These improvements are more noticeable as the input set becomes larger, as shown by the table below.

	Size of argument set				
	3	4	5	6	7
with memo	0.023138	0.113327	0.570707	3.13558	33.7762
without memo	0.03399	0.160506	0.982914	6.92302	130.645

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). As a result, it requires only a lookup which is a constant-time operation since the argument to `reach` is a constant. As the following data shows, the overhead of a lookup is minimal and the running time grows linearly with the number of nodes.

	# of nodes					
	4	8	12	16	20	25
graph1	0.00546	0.01253	0.0214	0.03563	0.05238	0.07923
graph2	0.00505	0.00920	0.0198	0.03236	0.05321	0.07524

In the above table, all the nodes of *graph1* are arranged in a single cycle. Every call to `reach` is memoized, and there will be a single lookup. On the other hand, *graph2* is a binary tree (an acyclic graph), and therefore it does not require memoization for correct execution, but still every call on `reach` is memoized by the implementation. Although figures for *graph1* are slightly higher, its performance is comparable to that of *graph2*. These examples show that memoization does not degrade the overall performance. Below we show the results of executing the `reach` program with a fully connected graph, where the number of edges is $n \times (n-1)$ and n is the number of nodes in the graph. Once again, as the following data shows, its performance is similar to the above result, in that its running time is linear in the number of edges.

# of nodes (# of edges)				
4(12)	5(20)	6(30)	7(42)	8 (56)
0.01017	0.01729	0.02792	0.04431	0.06327

4.2.2 Performance of Re-Do

We present two different examples to illustrate the performance of re-do. Our first is a formulation of the shortest-distance example in two different ways: with and without re-do. 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. The use of memoization in this example is exclusively for efficiency reasons.

```
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 below as `short2`:

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

The execution of `short2` requires memoization and re-do. The two functions were run 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 number in parentheses in the row for `short1` refers to the value of `L` that was used in making the top-level call (the smallest value needed to compute the correct answer was used). The number in parentheses in the row for `short2` refers to numer of redo's that were performed.

	# of nodes (# of edges)			
program	8(18)	16(38)	24(58)	32 (78)
`short1`	0.016637(2)	0.242124(6)	0.932096(10)	2.42102(14)
`short2`	0.050776(3)	0.2083(6)	0.463854(9)	0.856015(12)

Our second example shows the performance of memoization and re-do for the company controls example from section 2.2.2:

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

The following figures are obtained by running the `controls` function on extensional databases of different size, and observing the effect of the number re-do operations on the overall execution time.

# of edges	*# of Re-do's*	*Time*
8	2	0.030011
13	3	0.059156
18	4	0.09902
23	5	0.149445

The term *# of edges* in the above table 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. The above figures indicate to us that re-do is an efficient means of performing monotonic aggregation.

5. 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, or 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 [War83] 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, lazy, or memoized modes. The compiled code for every subset clause ends with a `collect` instruction, which tests the mode register in order to return a result that is appropriate to the calling mode.

- 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), lookup points (which identify the place where a re-do is required), and lazy choice points (to protect the lazy search space from normal backtracking). 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.

We have mentioned two opportunities for static analysis: detection of the property of distribution over union, and detection of lazy evaluation. 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 examining ways to automatically detect when to lazily enumerate the set elements. Detection of this property necessitates a more global analysis of the use of the lazy set. We are also 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

The implementation of SuRE was carried out by Kyonghee Moon as part of her Ph.D. dissertation, and is being made available by anonymous FTP from `ftp.cs.buffalo.edu`. This research was supported in part by grant DCR 9004357 from the National Science Foundation.

REFERENCES

[Ait91] H. Ait-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*, The MIT Press, 1991.

[AG91] S. Abiteboul and S. Grumbach, A Rule-Based Language with Functions and Sets, *ACM TODS*, 16(1):1–30.

[CG86] K. Clark and S. Gregory, Parlog: Parallel Programming in Logic, *ACM TOPLAS*, 8(1):1–49, 1986.

[DP93] A. Dovier and E. Pontelli, A WAM-Based Implementation of a Logic Language with Sets, *Proc. PLILP*, Springer Verlag, August 1993.

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

[BNST91] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur, Set Constructors in a Logic Database Language, *JLP*, 10:181–232, 1991.

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

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

[GSZ93] S. Greco, D. Sacca, and C. Zaniolo, Dynamic Programming Optimization for Logic Queries with Aggregates, *Proc. ILPS 93*, pp. 575-589, MIT Press, 1993.

[Jan94] D. Jana, Semantics of Subset-Logic Languages, Ph.D. dissertation, Department of Computer Science, SUNY-Buffalo, August 1994.

[Jay91] B. Jayaraman, The SuRE Programming Framework, TR 91-011, Department of Computer Science, SUNY-Buffalo, August 1991.

[Jay92] B. Jayaraman, Implementation of Subset-Equational Programs, *JLP*, 11:299-324, 1992.

[JN88] Jayaraman, B. and Nair A., Subset-Logic Programming: Application and Implementation, *Proc. JICSLP*, pp. 841-858, Seattle, 1988.

[JP87] B. Jayaraman and D.A. Plaisted, Functional Programming with Sets, *Proc. Third FPCA*, pp. 194-210, Springer-Verlag, 1987.

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

[JJ94] D. Jana and B. Jayaraman, Set Constructors, Finite Sets, and Logical Semantics, submitted to *JLP*. Available as TR 94-030, Department of Computer Science, August 1994.

[JOM95] B. Jayaraman, M. Osorio, and K. Moon, Partial Order Programming (Revisited), *Proc. Algebraic Methodology and Software Technology (AMAST 95)*, Springer Verlag, July 1995 (to appear).

[Kup90] G. M. Kuper, Logic Programming with Sets, *JCSS*, 41(1):44–64.

[KS91] D.B. Kemp and P.J. Stuckey, Semantics of Logic Programs with Aggregates, *Proc. ILPS 91*, pp. 387-401, MIT Press, 1991.

[Llo87] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1987.

[MPR90] I.S. Mumick, H. Pirahesh, and R. Ramakrishnan, The Magic of Duplicates and Aggregates, *Proc. 16th VLDB*, pp. 264-277, 1990.

[OJ93] M. Osorio and B. Jayaraman, Subset Assertions and Negation-as-Failure, *Proc. ILPS 93*, poster paper, MIT Press, 1993. Full paper available as TR 93-027, Department of Computer Science, SUNY-Buffalo.

[Prz88] 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.

[RS92] K.A. Ross and Y. Sagiv, Monotonic Aggregation in Deductive Databases, *Proc. 11th PODS*, pp. 114-126, ACM Press, 1992.

[SSRB93] S. Sudarshan, D. Srivastava, R. Ramakrishnan, and C. Beeri, Extending the Well-Founded and Valid Semantics for Aggregation, *Proc. ILPS 93*, pp. 590-608, MIT Press, 1993.

[SW94] T. Swift and D.S. Warren, Analysis of SLG-WAM Evaluation of Definite Clause Programs, *Proc. ILPS 94*, pp. 219-235, MIT Press, 1994.

[Van92] A. Van Gelder, The Well-Founded Semantics of Aggregation, *Proc. 11th PODS*, pp. 127-138, ACM Press, 1992.

[War83] D.H.D. Warren, An Abstract Instruction Set for Prolog, Technical Note 309. SRI International, 1983.

[War92] D.S. Warren, Memoing for Logic Progams, *CACM*, 35(3):93-111, 1992.