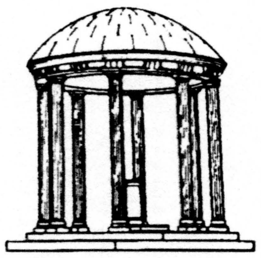
Compilation of Subset-Logic Programs

TR88-047

September 1988

Anil Nair

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



Compilation of Subset-Logic Programs

by

Anil Nair

A Thesis submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

1988

Approved by:

[illegible]

Advisor – Dr. Bharat Jayaraman

David A. Plaisted

Reader – Dr. David Plaisted

Jan F. Prins

Reader – Dr. Jan Prins

ANIL NAIR: Compilation of Subset-Logic Programs (Under the direction of Dr. BHARAT JAYARAMAN).

Abstract

Subset logic programming is a paradigm of programming with subset and equality assertions, and whose execution model is based on associative-commutative (a-c) matching and innermost reduction. SEL (Set Equational Language) is a language that has been proposed to illustrate this approach. This thesis describes the design and implementation of a system that compiles SEL programs into an instruction set similar to the Warren Abstract Machine instructions for Prolog. The novel aspects of our implementation include the compilation of a-c matching, backtracking upon failure and success, and the implementation of quantifiers over sets. This implementation has been completed and will run under any Unix system.

Acknowledgements

I would like to thank Dr. Jayaraman for his tireless and excellent advice and overall support throughout my work. Without his help and guidance this thesis could not have been done.

I would like to thank Frank Silbermann for reading the thesis and suggesting improvements. Thanks also to Gopal Gupta for all the tidbits of information which made the implementation easier.

Thanks to Alice, too.

To Amy

Contents

1 Introduction

Logic programming has gained a lot of popularity in the last decade largely due to its declarative style of programming and the success of Prolog. Prolog is based on predicate logic programming, and its success has made the term "logic programming" to be synonymous with "predicate logic programming." However, in recent years researchers have experimented with new forms of logic programming, notably constraint logic programming [JL87] and equational logic programming [O85]. In this thesis, we explore yet another approach, called subset logic programming [JP87,JN88].

The primary motivation for subset logic programming was to give a correct and efficient basis for programming with sets. Although set constructs are to be found in existing functional and logic languages (e.g., the "setof" construct of most Prolog systems [N85] and the relative-set construct in Miranda [T85]), these constructs do not support true sets. These languages have sacrificed clean semantics in incorporating sets as extra features.

The practicality of predicate logic programming has been enhanced by recent advances in compilation [W83], i.e., techniques for compiling unification and the control strategy of Prolog (depth-first search with backtracking). Compilation has speeded up the execution of predicate logic programs by at least an order of magnitude over interpreting.

This thesis presents similar efficient ways of compiling subset logic programs. We describe the implementation of a subset logic language called SEL (Set Equational Language) [JP87,JN88]. A SEL program consists of equality and subset assertions, and is executed using innermost reduction and restricted associative-commutative matching. These features are compiled into an instruction set similar to the Warren Abstract Machine (WAM) used for Prolog [W83].

The rest of this document is organized as follows. Chapter 2 describes the SEL language informally and gives a few examples. Chapter 3 describes the SEL system that has been implemented and guidelines for using it efficiently. Chapter 4 describes the two phases of the implementation: compilation to WAM-like instructions and emulation of the instruction set by software. Chapters 2 and 3 should serve as a users' manual for the system, while Chapter 4 should serve as an implementors' manual. Chapter 5 presents a brief summary and directions for further work.

We assume that the reader has some familiarity with Prolog terminology (e.g., unification) and the basic implementation issues involved in Prolog.

2 SEL: The Language

This chapter gives the syntax and informal semantics of SEL programs, along with examples.

2.1 Syntax

In the following BNF description of the syntax for SEL, we use the typewriter font for keywords and literals.

rules ::= *rule* | *rule rules*

rule ::= *equation* `.` | *subset* `.`

equation::= *function*`(` *terms* `)` `=` *expr*

subset ::= *function*`(` *terms* `)` `contains` *expr*

terms ::= *term* | *term* `,` *terms*

term ::= *boolean* | *integer* | *atom* | *variable* | *listterm* | *setterm*

listterm ::= `[ ]` | `[` *term* `|` *term* `]` | `[` *terms* `]`

setterm ::= `{ }` | `{` *term* `|` *term* `}` | `{` *terms* `}`

expr ::= *term* | *function* `(` *exprs* `)` | `if` *expr* `then` *expr* `else` *expr*

exprs ::= *expr* | *expr* `,` *exprs*

The data objects of SEL are booleans, integers, atoms, lists, and sets. The current implementation of SEL provides only integer numbers, e.g. `10`, `-3999`, etc; real numbers are not supported. The booleans are `true` and `false`. Any sequence of characters enclosed within single quotes is taken to be an atom, e.g., `'apple'`, `'also an atom'`, etc. Note that SEL does give significance to the case of the alphabet, e.g., the atom `'a'` is different from `'A'`. A variable is any sequence of alphanumeric characters starting with an alphabet. It could start with upper or lower case (unlike Prolog), e.g., `index`, `ELEMENT` etc. A list `[1,2,3]` is syntactic sugar for `[1 | [ 2 | [ 3 | [ ] ] ] ]`. The list pattern `[h | t]` is used to match the head and tail of a list, as in Prolog. Similarly a set `{1,2,3}` is syntactic sugar for `{1 | { 2 | { 3 | { } } } }`. A set pattern `{x | t}` matches a set by matching `x` against one element of the set and `t` against the rest of the set. The "_" symbol represents the "don't care" variable, as in Prolog.

2.2 Informal Semantics

SEL programs consist of equality and subset assertions. At the top level the user enters an expression, referred to as the **goal**. The goal has to be a ground expression, i.e., an expression with no variables in it. Its value is obtained by reducing the expression in leftmost innermost order. Since arguments to functions must be ground terms, function application requires one-way matching. rather than unification. The matching operation is actually a restricted form of associative-commutative matching [JN88].

Associative commutative matching [P72] can be used to match ground terms (possibly sets) against patterns (possibly set patterns). For example, if a set {1,2,3} is matched against the pattern {h|t}, it produces three matchings, viz., {h $\leftarrow$ 1. t — {2,3}}, {h $\leftarrow$ 2, t $\leftarrow$ {1,3}}, and {h $\leftarrow$ 3, t $\leftarrow$ {1,2}}. Note that all set patterns are of the form { $term_1$ | $term_2$ } rather than the more general $term_1 \cup term_2$. Patterns of the form x $\cup$ y are useful in iterating over all *subsets* of a set, but they are computationally expensive and do not occur frequently in practice; hence we do not support them. Patterns of the form {x | t}. on the other hand are useful for iterating over all *elements* of a set and are needed often. The complete matching algorithm is described by the following Prolog program. The first argument of `match` is a possibly non-ground term, representing the head of an assertion, and the second argument is a ground term, representing the arguments of a function call.

```
match(A,A) :- atomic(A), !.
match([],[]).
match({},{}).
match(V,Arg) :- var(V), !, V = Arg.
match([T1|T2], [Arg1|Arg2]) :-
              match(T1,Arg1), match(T2,Arg2).
match({Elem1|Set1},Argset) :-
              generate(Argset,Elem2,Set2),
              match(Elem1,Elem2),
              match(Set1,Set2).
```

```
generate({Elem|Set},Elem,Set).
generate({Elem|Set},Elem2,{Elem|Set2}) :-
                    generate(Set,Elem2,Set2).
```

It is important to note that we do not follow this recursive algorithm literally in our implementation. The main purpose of compilation is to avoid the general matching algorithm in the simpler cases.

Now we describe the meaning of equality and subset rules. If an innermost expression matches an equality rule, it is replaced by the body of the rule (r.h.s.) after substituting for the variables on the left hand side suitably. Note that if more than one equality rule matches or if one equality rule matches in more than one way, any one match is used to reduce the body of the rule. Thus, the programmer has to make sure that the result is independent of which match is chosen. If the expression matches a subset rule, the right hand side is reduced for each different a-c match, and the expression is reduced to the union of the sets obtained for each a-c match. If none of the subset rules match, the expression is reduced to the null set. If more than one subset rule matches, the expression is reduced to the union of the right hand sides of all matching rules. This behaviour follows from the *closed world* assumption of SEL, i.e., a set is completely defined by its subsets; there are no other elements in the set than the ones specified.

A non-set valued function is undefined (denoted by ?) if there are no equality rules defining it or if none of the equality rules match. The undefined set, on the other hand, is the empty set. We also define {?} = {}. That is, an undefined value as an element of a set can be dropped from the set. This captures the notion of "emptiness as failure".

The conditional expression `if` e_1 `then` e_2 `else` e_3 reduces to e_2 if e_1 reduces to `true`, and to e_3 if e_1 terminates but is not `true`. That is, the conditional captures a form of "negation by failure".

2.3 Examples

The standard LISP-like definition of the "append" of two lists can be defined using the following equality rules.

```
append([],y) = y.

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

The following assertions illustrate the use of subset rules to define the cross-product and intersection of two sets. Note that no assertion is needed when one of the argument sets is empty. The result is the empty set in these cases, by the closed world assumption discussed earlier.

```
crossproduct({x|_},{y|_}) contains {[x|y]}.

intersect({h|_},{h|_}) contains {h}.
```

The "permutations" example below illustrates recursive subset rules. Note that perms takes a set as argument and returns a set of lists.

```
perms({ }) = {[ ]}.

perms({x|t}) contains distr(x,perms(t)).

distr(x,{y|_}) contains {[x|y]}.
```

The "8-queens" program below shows how SEL can be used to formulate fairly complex problems easily. The queens function, when invoked at the top level as queens(1,{}) returns the set of all solutions to the 8-queens problem.

```
queens(col,safeset) = if  eq(col,9)
                          then {safeset}
                          else placequeen(col,{1,2,3,4},safeset).
placequeen(col,{row|_},safeset) contains
                       if  safe([col|row],safeset)
                           then queens(col+1,{[col|row]|safeset})
                           else {}.
safe([c1|r1],{}) = true.
safe([c1|r1],{[c2|r2]|s}) = (abs(c1 - c2) <> abs(r1 - r2))
                            and (r1 <> r2) and safe([c1|r1],s) .
```

2.4 Quantifiers over sets

SEL provides the **ifall** and **ifone** constructs in order to simulate quantifiers over sets. For example, the predicate

$$p(s) = (\forall x \in s)q(x) .$$

can be defined in SEL as

```
p({x|_}) ifall q(x).
```

Similarly, the predicate

$$p(s) = (\exists x \in s)q(x)$$

can be defined as

```
p({x|_}) ifone q(x).
```

Operationally, the **ifall** rule says that the predicate being defined is true if for all a-c matches of the head, the body of the rule reduces to **true**. The **ifone** rule says that the predicate is true if for any one match the body of the rule reduces to **true**. For example, we can define the function **disjoint** of two sets using the following definition.

```
disjoint({x|_},{y|_}) ifall x <> y.
```

The default cases of the **ifall** and **ifone** rules should be noted. When a **ifone** rule defines a predicate and there are no matches for the rule, the default returned value is **false**. But when a **ifall** rule defines a predicate and there are no matches for the rule, then the default returned value is "true". The current implementation also allows a predicate to be defined using multiple **ifall** rules or multiple **ifone** rules. When multiple **ifall** rules define a predicate, execution chains through them until one of the right hand sides reduces to a non-true value. However, when multiple **ifone** rules define a function, execution chains through them until one of the right hand sides reduces to **true** in which case the result is **true**. Note: It is illegal to use both **ifall** and **ifone** rules in defining a predicate, because the default for the **ifall** conflicts with that of the **ifone** rule.

2.5 Distribution over Union

We now discuss an important property of certain SEL functions, which has implications for considerable performance improvement. A function f is said to distribute over union in some argument iff

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

There are some important benefits if we know which functions have this property. For example, in computing $f(\ldots, g(\ldots), \ldots)$ we need not compute the entire set that $g(\ldots)$ stands for; instead we can compute one element of $g(\ldots)$ at a time and apply f to each one of these singleton sets and propagate the union. We save time by computing one element at a time because we avoid checking for duplicates of the intermediate set. We also save space because we avoid accumulating a possibly large intermediate set. (This optimisation is similar to the transformation in functional languages that avoids the creation of intermediate lists.) In the current system, we assume that the programmer annotates programs indicating which functions distribute over union in which argument. For example, in the **perms** definition of the previous section, the **distr** function distributes over union in its second argument. This can be annotated as

```
distr(x,{y|_}) contains {[x|y]}.  distribute(distr,2).
```

The **distribute** annotation must appear after the definition of the function. In the current implementation, multiple **distribute** annotations must be used to specify that a function distributes over union in more than one argument.

When a function is called to produce one subset at a time rather than the entire set, we say that the function is called in "call-one" mode. When it is called to return the entire set, we say the function is called in "call-all" mode. In the permutations example, we find that the **perms** function can be invoked in "call-one" mode. Although there appears to be a big advantage of using call-one, in this case it turns out that the advantage is not much because of the following trade-off. If we call **perms** in call-one, each call to **distr** will distribute a constant over a one element set, resulting in many more calls to **distr** compared with invoking **perms** with call-all mode. Thus the cost of extra function calls nearly equals the cost of checking for duplicates and forming intermediate sets. In this example, there are $n * n!$ calls to

distr if it is called in call-one mode, whereas there are only $n + n * (n-1) + \ldots + n!$ calls to **distr** if it is called in call-all mode. For a 5 element set, this means that there are $5 * 5! - (5 + 5 * 4 + \ldots + 1) = 600 - 325 = 275$ calls more calls to **distr**. A trace of call-one against call-all for the goal **perms({1,2,3})** is given below to clarify this point. Note that **distr** is always called with a singleton set in the left column.

perms({1,2,3}) with distribute(distr,2)	perms({1,2,3})
Call to perms({1,2,3}).	Call to perms({1,2,3}).
Call to perms({2,3}).	Call to perms({2,3}).
Call to perms({3}).	Call to perms({3}).
Call to perms({}).	Call to perms({}).
Call to distr({[]},3).	Call to distr({[]},3).
Call to distr({[3]},2).	Call to distr({[3]},2).
Call to distr({[2,3]},1).	Call to perms({2}).
Call to perms({2}).	Call to perms({}).
Call to perms({}).	Call to distr({[]},2).
Call to distr({[]},2).	Call to distr({[2]},3).
Call to distr({[2]},3).	Call to distr({[3,2],[2,3]},1).
Call to distr({[3,2]},1).	Call to perms({1,3}).
Call to perms({1,3}).	Call to perms({3}).
Call to perms({3}).	Call to perms({}).
Call to perms({}).	Call to distr({[]},3).
Call to distr({[]},3).	Call to distr({[3]},1).
Call to distr({[3]},1).	Call to perms({1}).
Call to distr({[1,3]},2).	Call to perms({}).
Call to perms({1}).	Call to distr({[]},1).
Call to perms({}).	Call to distr({[1]},3).
Call to distr({[]},1).	Call to distr({[3,1],[1,3]},2).
Call to distr({[1]},3).	Call to perms({2,1}).
Call to distr({[3,1]},2).	Call to perms({1}).
Call to perms({2,1}).	Call to perms({}).
Call to perms({1}).	Call to distr({[]},1).

Call to perms({}).

Call to distr({[]},1).

Call to distr({[1]},2).

Call to distr({[2,1]},3).

Call to perms({2}).

Call to perms({}).

Call to distr({[]},2).

Call to distr({[2]},1).

Call to distr({[1,2]},3).

Call to distr({[1]},2).

Call to perms({2}).

Call to perms({}).

Call to distr({[]},2).

Call to distr({[2]},1).

Call to distr({[1,2],[2.1]},3).

The following table shows the time taken by the two approaches for all permutations of a 3.4.5, and 6 element set. All time measurements are in milliseconds.

	Number of elements in the set			
	3	4	5	6
call-all	42	183	1500	27300
call-one	42	166	1350	23200

It turns out that the call-one mechanism provides a limited form of lazy evaluation. To understand this point, we must examine `call-one` further. When a function is called using **`call-one`**, it computes one subset of the solution, suspends the other matches. and returns to the caller. When resumed later (due to failure or success), the suspended computation produces a new subset derived from the next a-c match.

Call-one can also be used to implement a form of "generate and test." Suppose, in the program below, the function `test` fails if its argument set does not satisfy some conditions. Then we can define `test` as distributing over union, and `test` can act on one element of `generate` at a time. If none of the elements produced by `generate` is accepted by `test`, then `test` returns undefined.

```
generate(...) contains ....

test({x|_},...) = ....
```

```
distribute(test,1).
```

SEL does not support infinite objects. But we can use the control behaviour of `call-one` as a programming trick to define functions that potentially accept infinite sets as arguments. Below, function `f` works element-at-a-time, and hence it can accept an infinite set as an argument. The details of predicate `p` are not relevant here.

```
natural(x) contains {x}.

natural(x) contains natural(x+1).

f({y|_}) = p(y).

distribute(f,1).
```

For example, if the top-level goal is `f(natural(10),...)`, `natural` will generate subsets {10}, {11}, ... until we get one on which `p` will not fail.

Note that in the previous two examples, the functions `f` and `test` are not set-valued functions and they do not truly have the property of distributing over union, but they do "distribute" in a more general sense.

2.6 Avoiding Check for Duplicates

Checking for duplicates is the most time consuming operation when computing with sets in a programming language. In many practical uses, the sets defined by multiple subset assertions are found to be disjoint. In these cases, SEL allows the programmer to request bypassing the check for duplicates by using annotations. For example, the definition for `product` of two sets does not produce duplicates, and hence can be annotated to prevent checking for duplicates as follows:

```
product({x|_},{y|_}) contains {[x|y]}.

nodup(product).
```

The annotation `nodup(product)` may appear anywhere *after* the definition of product. Among the examples in the previous chapter, functions `intersect`. `queens`, `perms`. `distr` do not generate duplicates, and could each be annotated with the "nodup" clause. A performance comparison for computations with and without duplicate checking is given in the following table. All times are in milliseconds.

goal	With duplicate checking	Without duplicate checking
perms of a 5 element set	1500	916
all solutions to the 8 queens problem	78000	75000
product of two 10 element sets	416	133
the power set of a 6 element set	350	96

2.7 Error Trapping

The "emptiness as failure" notion provides a limited way to trap errors. For example, a function `f(x)` that returns ? (undefined) on certain inputs can be augmented to give an error message in the following way:

```
trap(x) contains {f(x)}.
action({},x) = ['bad','input',x].
action({x},_) = x.
toplevel(x) = action(trap(x),x).
```

Now, **`toplevel(x)`** would return the same value as **`f(x)`** if **`f(x)`** is defined, but would return an error message if **`f(x)`** is not defined. Note, that general exception handling is much more powerful.

3 The User Interface

This chapter describes the user interface and special features of the current implementation, and offers some hints for debugging and writing more efficient programs.

3.1 Entering and Leaving the System

To invoke the system, type **sel** along with any flag options from the Unix shell (see appendix A). The interpreter responds as follows:

```
SEL Version 1.0
sel>
```

Upon receiving the prompt, the user can type in any goal terminated with a period and carriage return. For example the query

```
sel> 3 * 4.
```

results in

```
12
sel>
```

To exit the interpreter, type CTRL-d at the prompt. The interpreter responds with

```
SEL Execution halted
```

and exits back to the shell. To abort a run-away computation or to suspend SEL, use the regular shell kill characters. SEL does not trap any of these interrupts. A sample of a complete session with the system is given in appendix B.

3.2 Compiling Files

The only way to access and execute user-defined functions in SEL is by placing the definitions in a file and compiling them with the **compile** command. For example. to execute the **append** function, enter its definition in a Unix file, say **append.sel**. and compile it as follows.

```
sel> compile('append.sel').
```

SEL will respond with a list of functions that are now defined,

```
[append]
sel>
```

Now the **append** function can be invoked by typing

```
sel> append([1,2],[3,4]).
```

SEL responds with

```
[1,2,3,4]
sel>
```

In the current implementation, there is no facility to redefine functions or to type in rules interactively. The only way to do this is to leave the system and to edit the files containing the definitions and by starting the system all over again. The user can keep definitions in multiple files and `compile` these files as needed.

The user can optionally type an optimise flag after the filename to get SEL to compile set patterns, so that sets are adjusted in O(n) space and time, rather than making n different copies of the remainder of an n-element set (see section 4.1. para. 6). For example, **compile('foo',$opt)** would result in all set patterns in the file `foo` to be compiled with this optimisation.

3.3 Obtaining One Solution

As discussed in the previous chapter, a function that distributes over union calls its argument function using **call-one**. The **any** system call can be used if the user wants to call the top-level goal using **call-one**. If the function is defined using a subset rule, it would return to the top level after one subset is computed, rather than try all a-c matches. For example, after compiling the **perms** example, we can call the goal **perms({1,2,3})** using **call-one** by typing

```
sel> any(perms(1,2,3)).
```

The response of the system would be

```
{[1,2,3]}
```

3.4 Timing Goals

To time a goal, we provide the built-in function **cputime**. It returns the number of milliseconds used by cpu for the SEL process so far. For example, the goal `rev([1,2,3,4,5,6,7])` can be timed the following way.

```
sel> cputime()
50
sel> rev([1,2,3,4,5,6,7]).
[7,6,5,4,3,2,1]
sel> cputime()
66
```

This means that the goal took 66 - 50 = 16 ms to execute.

3.5 Error Handling and Tracing

Error handling is rather primitived in SEL. All lexical errors and syntactic errors are caught by the parser, which has been written using the Unix tools, "lex" and "yacc". The parser reports the line numbers on which errors occurred and the token near which the error occurred. On encountering an error, the entire rule or goal in which the error occurred is discarded. The most common syntax errors are due to using keywords as variable or function names. The following the keywords are reserved in SEL: `or`, `and`, `not`, `lessp`, `greaterp`, `numberp`, `listp`, `gretereq`, `lesseq`, `eq`, `neq`, `null`, `atom`, `plus`, `minus`, `times`, `divide`, `mod`, `div`, `abs`, `if`, `then`, `else`, `true`, `false`, `compile`, `contains` , `ifall`, `ifone`, `nodup`, `distribute`, `any`, `cputime`, `trace`.

In order to assist debugging, SEL has a trace feature that will show the calls being made along with the arguments. The only way to see the results of the function call is to see where the result of that function call is used. If it is passed as an argument to another function, look at the arguments of that function call. Tracing can be switched on by typing

```
sel> trace.
```

Selective tracing can be specified by providing the name of the function as an argument to `trace`. For example, to see the calls to some two functions `f` and `g`, but not any other functions type

```
sel> trace(f).
sel> trace(g).
```

To switch off tracing once again type

```
sel> trace.
```

3.6 Hints on Programming

We first offer a few hints on making SEL programs a little more efficient. The compiled code does clause indexing among multiple definitions of a function based on the first argument. For such a function, it is preferable, if possible, to rearrange arguments so that they differ in the first argument.

The two kinds of set-patterns {x|y} and {x|_} work with different efficiencies. If a function does not need the remainder of a set, use {x|_}, rather than {x y}, as {x|_} avoids constructing the remainder of the set.

The current implementation is not true to the semantics of SEL in a few respects. The behaviour `{?} = {}` is achieved only if the set-valued function is defined using subset rules. If an equality rule defines a set-valued function and an element of the set turns out to be undefined (?), then the function returns ?, rather than the set with all the remaining elements. The current implementation is also not completely correct with respect to the semantics of **`if-then-else`**. If the condition evaluates to `?`, the conditional expression reduces to ?, rather than to the value of the else clause.

The compiler does not check for the confluence of program assertions, and hence the order in which the equality assertions are placed is significant.

4 Execution of SEL

This chapter describes the abstract machine model used in implementing SEL, We assume that the reader has some familiarity with implementation issues in Prolog and the WAM (Warren Abstract Machine)[W83].

4.1 Flattening Expressions

The execution of SEL programs can be separated into two steps: compilation, followed by interpretation of the compiled code. Before compilation. all expressions are flattened to reflect leftmost innermost reduction order. This converts the body of each rule to a series of function calls. For example, shown below are the flattened forms of the **append** and **perms** definitions of Chapter 2.

```
append([ ], y) = y.
append([h|t], y) = [h|t1] :- append(t,y) = t1.


perms({}) = {[ ]}.
perms({x|t}) contains v1 :-  perms(t) contains v2,
                             distr(x , v2) = v1.
```

Note that **contains** is used in flattening **perms** in the r.h.s because **distr** distributes over union in its second argument. A nested SEL definition of the form

```
f(x) = g(h(i(x)))
```

gets flattened into a series of goals

```
f(x) = z :- i(x) = w, h(w) = y, g(y) = z.
```

The result of each function call can viewed as an extra argument of a Prolog-like definition. For example, the above may be viewed as

```
f(x,z) :- i(x,w), h(w,y), g(y,z).
```

4.2 Basic Execution Model

A function application is initiated by a `call` instruction. It is first matched against all equality rules defining the function. If there is a successful match and if the matching equality rule has *permanent* variables, an *environment* record is created on the *control stack* for the call. Control then transfers to the execution of the body of the equality rule. Once the body is executed, this rule is exited and control goes back to the caller of this function. If there are no equality rules defining the function or if none of them matches, then we try to match this call against any subset rules defining the function. If none of the rules match, *failure-backtracking* is initiated. The multiple subset rules that match a given call and the multiple a-c matches within any subset rule are attempted sequentially in a depth-first order. When entering a subset rule, a *choice point* is created on the control stack to keep track of rules not yet tried. A *choice point* is also saved when a subset rule matches, so as to try all possible matches within the rule. In this case, the choice point can have multiple branch points, one to record each occurrence of a set pattern in the head. (Note: this choice point is created only if the body of the call involves a function call, otherwise the rule is executed like local nested loops.)

As discussed in Chapter 2, a function can be called in two ways: `call-one` or `call-all`. If a subset assertion is called using `call-all`, each successful completion of the rule causes *success-backtracking* to the most recent choice point. If it is called using `call-one`, each successful completion causes an exit to the caller. The environment record is not deleted at this time. Once all branch points have been exhausted, the next subset assertion is attempted and the current environment is deleted. As each subset is computed it is added to the overall set after removing duplicates.

The actions on success and failure backtracking are nearly the same. The most recent choice point is retrieved and computation resumes from the information in the choice point. The effect of the backtracking is to produce the next subset. The difference between failure and success backtracking is that the subset computed from the current path is considered empty and hence neglected in the case of failure backtracking, whereas it is collected in the case of the success backtracking.

The control behaviour of `ifall` and `ifone` rules is a little different from that of the subset rules. If there are no equality rules, or if none of the equality rules match, then the `ifall` and `ifone rules` would be tried sequentially in a depth-first fashion. The difference from a subset rule is that if the body of a `ifone` (resp. `ifall`) reduces to a value `true` (resp. other than `true`) then control returns to the caller. The remaining matches and the remaining rules are not tried.

The instructions used to compile the various cases in a-c matching are similar to the ones used in WAM for unification. One of the main differences is that we can identify the terms that are going to be bound at compile time. This means that we can identify the *read* and *write* modes of WAM's *unify* instructions. We use the `match` and `store` instructions for the two cases respectively. The different actions taken by the WAM's `get` instructions when the argument is bound and unbound can also be recognized at compile time. We use the `get` and `store_indirect` instructions for the two cases. Note that the only unbound argument is the result of the function call, and the `store_indirect` instructions are used to return the result of function calls.

In order to deal with sets in a-c matching, we have introduced instructions to match sets against set patterns and to adjust the sets to produce different matches. The | set constructor is represented as a cell of two pointers, one to the head of the list and one to the rest of the cell. This is the structure that has to be "adjusted" to provide the different representations. There are three ways of adjusting sets. For example, if a set `{1,2,3}` pointed to by register `A1` is matched against `{h|_}`, after the first match (where `h` gets `1`), we can adjust `A1` to point to `{2,3}` (Fig. 4.1). This operation is performed by the `adj_set_head` instruction. If the set pattern is of the form `{h|t}`, we can construct the n remainders of an n-element set in O(n) space, using destructive modification (Fig. 4.2). Note that this adjust operation is a constant time operation. This can be done only in the case where `t` is not being returned (either directly or indirectly) as part of the answer. Detecting this case needs global data-flow analysis, which is not supported in the compiler yet. If the remainders do become part of the answer, we have to use the `adj_set_with_copy` instruction. This makes n copies for the n remainders and takes O(n^2) space. The `adj_set_with_copy` instruction is used by default for set patterns of the form `{x|t}` (see section 3.2).

```
adj_set_head Vi

1.  Vi  =  Vi.tail
```

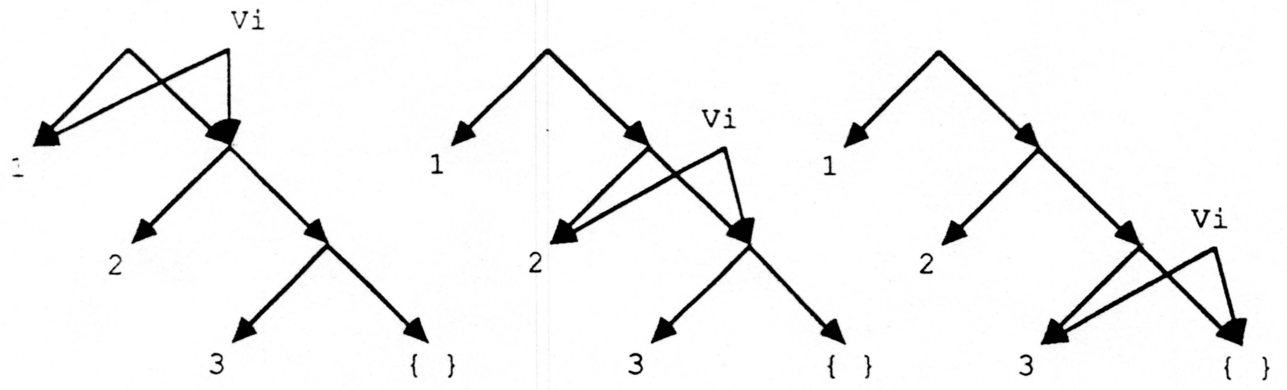


Fig. 4.1 adj_set_head

```
adj_set Vi

1. x = new node on heap
2. x.head = N.tail.head
3. x.tail = Vi
4. N.tail = N.tail.tail
5. Vi = x
```

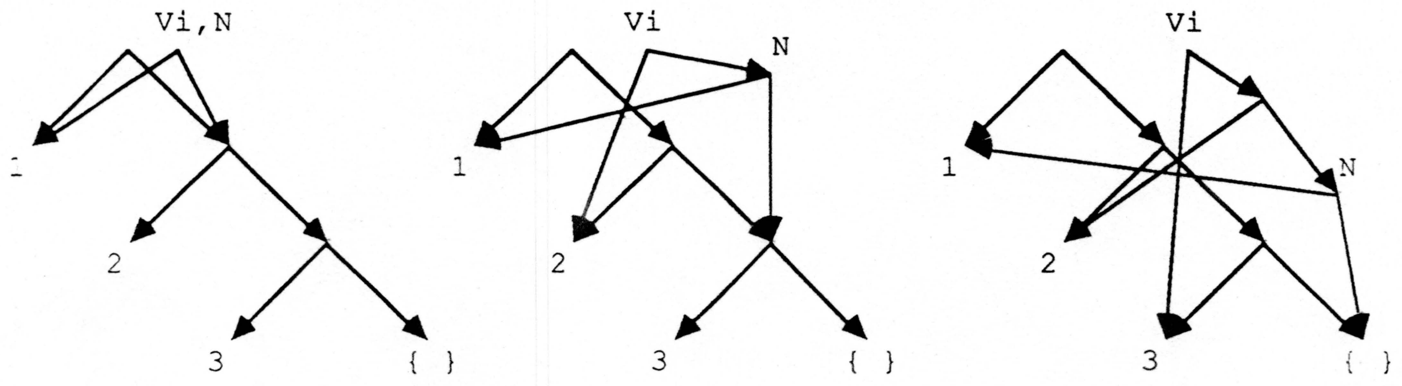


Fig. 4.2 adj_set

4.3 Data Objects

Every term in SEL is represented by a word containing a value and a tag. The types of data objects allowed presently include atoms. boolean values, integers, variables, lists and sets. Lists and sets are represented using a sequence of tagged pointer pairs, for the "head" and "tail" of the list or set.

4.4 Data Areas

The main data areas consist of the code area, the stack and the heap. The code area holds all the compiled code. The stack is used to allocate environments for function calls and choice points for backtracking. An environment allocated on the stack stores all permanent variables associated with a rule. It also has a continuation consisting of a continuation code pointer and a continuation environment pointer. A choice point is created if there are multiple subset rules to be tried or if a call is made as `call-one`. A choice point contains all information necessary to restore an earlier state of computation. The information stored includes a pointer to the other subset rule or the branch pointers `B1...Bm`. all argument registers `A1...An`, the continuation program pointer `CP`, `CE` the current environment pointer, `LCP` the last choice point and `M` the mode register. These are described in the next section in a little more detail. The heap is used to store all structured objects. Once structures are created on the heap, they are not retracted as is in the case in Prolog.

4.5 Registers

The registers that are used to store the current state of a SEL program in execution are:

P program pointer (to the code area)

CP continuation program pointer (to the code area)

CE current environment (on the local stack)

LCP last choice point (on the local stack)

H top of heap

S structure pointer (to the heap)

CB current branch point register

M mode register

A1,A2...An argument registers

X1,X2...Xn temporary registers

B1,B2...Bn branch pointers (to the code area)

As in the WAM, the `A` registers and `X` registers are identical: the different names merely reflect their different usages. The `A` registers are used to pass the arguments of a function call. The `X` registers are used to hold the values of temporary variables.

4.6 Compilation

SEL programs are compiled into instructions for an abstract machine. These instructions are then emulated in software. In general, each SEL symbol corresponds to one instruction. An instruction consists of an opcode with one, two or three operands. The opcode generally encodes the type of SEL symbol along with the context in which it occurs. An operand is either a constant. e.g.. an integer or atom, a register, or an address.

The entire instruction set can be divided into a number of classes, viz., the `get` instructions, the `store_indirect` instructions, the `match` instructions, the `put` instructions, the `store` instructions, the `procedural` instructions and the `indexing` instructions. We describe the compilation of SEL programs by describing each class of instructions.

The `get` instructions correspond to the terms in the head of a rule. and are responsible for matching the rule against the arguments of the function call, which are in the A registers. For example, if a `[]` in the head of a rule is to be matched against the third argument of a function call, it would get compiled into `get_nil A3`. The `get_variable` instruction is used for the first occurrence of a variable in the head. If it is not the first occurrence of the variable, the `get_value` instruction is used. The `get` instructions are:

```
get_variable Vn,Ai       get_value Vn,Ai
get_constant C,Ai        get_nil Ai
get_list Ai              get_set Ai,Vn
```

Note: Here and in the descriptions that follow, **Ai** represents an argument register, **Vi** represents a variable (which could be a temporary register **Xi** or a permanent variable **Yi**), and **C** represents a constant.

The **store_indirect** instructions are used to return the result of the function call, and is generated for the last argument of a function. The argument that is being matched against is sure to have a reference to an unbound variable. For example, if the result of a two-argument function call is the constant 5, then it is compiled to **store_ind_const 5,A3**. The **store_indirect** instructions are:

```
store_ind_nil Ai         store_ind_phi Ai
store_ind_const C,Ai
store_ind_list Ai        store_ind_set Ai
store_variable Ai        store_ind_value Ai
```

If the body of a rule is a variable (note: it cannot be the first occurrence), then the **store_ind_value** instruction is used. The **store_variable** instruction is used for all subset rules and for all equality rules with a function as the body of the rule.

The **match** instructions are used to match arguments of lists or sets in the head of a rule. This instruction is always preceded by a **get_list** or **get_set** instruction. For example, if a list pattern **[h|t]** is to be matched against the third argument of a function call, it would get compiled into

```
get_list X3              % [
match_variable Vi        % h |
match_variable Vj        % t ]
```

This example assumes it is the first occurrence of the variables **h** and **t**, and that there values are stored in variables Vi and Vj. Note: Here and in all following descriptions, we annotate the compiled code with the corresponding program text, as comments at the end of each line.

The `match` instructions are:

```
match_variable Vn        match_value Vn
match_nil                match_phi
match_const C
```

The `put` instructions are used to load the registers with the arguments before a function call is made. For example, if the second argument to a function call is the null set, then the corresponding compiled code for that would be `put_phi A2`. The `put` instructions are:

```
put_nil Ai               put_phi Ai
put_const C,Ai
put_variable Yn, Ai      put_value Vn, Ai
put_list Ai              put_set Ai
```

The `put_variable` instruction is used only to load the address of the place where the result of the function call should go.

The `store` instructions are used to load arguments of lists and sets. They follow either a `put_list`, `put_set`, `store_ind_list` or a `store_ind_set` instruction. For example, if the first argument to a function call is `[2|h]`, it would get compiled into

```
put_list A1              % [
store_const 2            % 2 |
store_value Vi           % h ]
```

The `store` instructions are:

```
store_nil                store_phi
store_const C
store_variable Vn        store_value Vn
```

The `store_variable` instruction is used only when the argument to the list or set constructor is a function call.

The **procedural** instructions are responsible for control transfer, environment allocation, and function invocation. The **procedural** instructions are:

```
allocate                    deallocate
execute P                   proceed
call_all P,N                call_one P,N
save_choice_point?          collect?  Vm,Vn
ifone Vm,Vn                 ifall Vm,Vn
if_false_jump Vn,Ptr        jump Ptr
```

where `P` represents the code pointer of a function, **N** is the number of variables (still in use) in the environment, and `Ptr` is a pointer in the code for an `if-then-else` statement.

The `allocate` instruction appears at the beginning of any equality rule that has two function calls in its body (in its flattened form), or any subset rule that has one function call. The `deallocate` rule appears before the call to the last function in an equality rule. In the case of the subset rule, the `collect` does the deallocation too. The `execute` instruction is used to call the last function in the body of an equality rule. This is how last-call optimisation is achieved. The `proceed` instruction ends an equality rule with no function calls in its body. All functions in the body of a subset rule and all functions in an equality rule except the last one is invoked with a `call` instruction. The `call-one` instruction is used if we have distribution over union. The `call-all` instruction is used for all other cases. The first argument for all these instructions is the pointer to the code of the function that is called. The second argument is the number of variables that are still in use in the current environment. This facilitates environment trimming. The `collect?` instruction appears at the end of each subset rule. The first argument is the destination of the result of the function call (the union of all subsets) and the second argument is where each subset gets stored. The `save_choice_point?` instruction appears after the head and before the body of each subset rule. The `ifone` and `ifall` instructions appear at the end of an ifone and ifall rule respectively. The `if_false_jump` and the `jump` instructions are used to compile the control of an `if-then-else` in the body.

The `indexing` instructions are used to index among the definitions for one function. The indexing instructions are:

```
try_equ_else P          try_sub_and P
switch_on_ground_term Lc,Ll,Ls
```

The **try_equ_else** instruction appears before every equality rule which has another equality rule. with the same type of first argument, following it. The **try_sub_and** instruction precedes the definition of every rule which has a subset rule, with the same type of first argument, following it. Note that all equality rules will be tried first and the sunset rules after. In both these case the argument is the code pointer for the next rule. The **switch_on_ground_term Lc,Ll,Ls** is used to do clause indexing. This instruction appears at the beginning of any function that is defined with multiple rules having different first arguments. **Lc**, **Ll**, **Ls** are the addresses of the definitions which have a constant, a list or a set as their first argument respectively.

4.7 Examples of Compiled Code

Now we present a few examples of programs with their complete compiled code. First we give the SEL definitions and follow it with the compiled code with each instruction commented with the corresponding symbol in the SEL program in its flattened form. The append and qsort examples should illustrate the basic use of most instructions.

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

```
    switch_on_ground_term L1,L2,fail
L1: get_nil A1                   % append ([],
    get_variable X4, A2          % y )
    store_ind_value X4,A3        % = y
    proceed                      % .
L2: get_list A1                  % append( [
    match_variable X4            % h |
    match_variable X5            % t ],
    get_variable X6, A2          % y )
    store_ind_list A3            % = [
    store_value X4              % h |
    store_variable X7           % z ]
    put_value X5, A1            % where append( t,
    put_value X6, A2            % y )
    put_value X7, A3            % = z
    execute append/2            % .
```

```
qsort([]) = [].
qsort([p|l]) = q2(p, partition(l,p)).
```

```
    switch_on_ground_term L1,L2,fail
L1: get_nil A1                  % qsort([])
    store_ind_nil A2            % = []
    proceed                     % .
L2: allocate
    get_list A1                 % qsort( [
    match_variable Y1           % p |
    match_variable Y2           % l ] )
    get_variable Y3, A3         % = z
    put_value Y2, A1            % where partition( l,
    put_value Y1, A2            % p)
    put_variable Y4, A3         % = y ;
    call_all partition, 4       %
    put_value Y1, A1            % q2( p,
    put_value Y4, A2            % y)
    put_value Y3, A3            % = z
    deallocate                  %
    execute q2                  % .
```

The intersect and perms examples should illustrate the use of set patterns and subset rules.

```
intersect({h|_},{h|_}) contains {h}.
```

```
    get_set A1, X4            % intersect( {
    adj_set_head X4           %
    match_variable X5         %     h |_ } ,
    get_set A2, X5            % {
    adj_set_head X5           %
    match_value X5            %     h |_ } )
    save_choice_point?        % ⊇
    store_ind_set X6          % {
    store_value X5            % h |
    store_phi                 % {} }
    collect?  A3, X5          % .
```

```
perms({}) = {}.
perms({h|t}) contains distr(perms(t),h).
distribute(distr,1).
```

```
    switch_on_ground_term L1,fail,L2
L1: get_phi A1                % perms()
    store_ind_set A2          % = {
    store_nil                 % [] |
    store_phi                 % {} }
    proceed                   % .
L2: get_set A1, Y1            % perms({
    adj_set Y1                %
    match_variable Y2         % h |
    match_variable Y3         % t }
    get_variable Y4, A2       % ⊇ v1
    save_choice_point?        % where
    put_value Y3, A1          % perms(t)
    put_variable Y5, A2       % ⊇ v2 ;
    call_one perms, 6         %
```

```
put_value Y5, A1            % distr( v2,
put_value Y2, A2            %  h)
put_variable Y6, A3         % ⊇ v3
call_all distr, 6           % .
collect? Y4, Y6             % v1 = v1 ∪ v3
```

4.8 The Instruction Set

This section describes the actions taken on executing each instruction. Note: In the following descriptions. Vn is generically used to denote a permanent variable Yn, or a temporary variable Xn. Instructions marked with an asterisk are similar to those of the WAM [W83].

4.8.1 Control Instructions

allocate* This instruction appears at the beginning of a rule that has at least one permanent variable. A frame is allocated on the top of the stack after the last choice point or environment. The continuation is saved in the new environment and the environment pointer CE is set to this frame.

deallocate* This instruction appears before the last call of a rule that has permanent variables. The previous environment is restored from the continuation and the current environment is discarded.

call_all Proc, N* This instruction appears if there is a function call on the right hand side of a rule. If the function call is an argument to another function and if that function distributes over union, then `call_one` is used instead of `call_all`. The continuation pointer CP is set to the following code and control is transferred by setting the program counter P to Proc. N is the number of variables in the current frame that may be used after this call. The mode register M, is set to the *allmode*.

call_one Proc, N This instruction is used to invoke a function that appears as an argument of another function that distributes over union in this argument.

collect? Vm, Vn This instruction occurs at the end of every subset rule. It has serves both control and data functions, which depend on the mode in which the rule was called. If the mode is *allmode*, it computes the union of **Vm** and **Vn** by assigning the tail of the set pointed to by **Vn** to **Vm**, and assigns the result to **Vm**. Control passes to the most recent choice point or to any branch point within this rule. If the mode is *onemode*, **Vm** is set to **Vn** and control goes back to the caller. This is done by setting the environment pointer **CE** and the program counter P from the continuation.

ifone Vm,Vn This instruction occurs at the end of every "ifone" rule. If the body of the rule reduced to a value other than **true** (**Vn** has a value other than **true**), control passes to the most recent choice point or to any branch point within this rule if it exists. If **Vn** has the value **true**, **Vm** is set to **true**, the current branch point CB is set to 0, and control goes back to the caller.

ifall Vm,Vn This instruction occurs at the end of every "ifall" rule. If the body of the rule reduced to a value **true** (**Vn** has a value **true**) then control passes to the most recent choice point or to a branch point within this rule if it exists. If Vn has a value other than **true** then Vm is set to **false**, the current branch point CB is set to 0, and control goes back to the caller.

execute Proc* This instruction makes the outermost function call of the body of an equality assertion. The program counter P is set to Proc.

proceed* If the right-hand side of a rule does not have a function call, then it is terminated with this instruction. The program counter P is set to the continuation pointer CP.

save_choice_point? This instruction appears after the head of every subset rule. Its action is also dependent on the mode in which the rule is called. If the rule was called in *allmode* then no choice point is created. If the rule was called in *onemode* then a new choice point is created and all registers A1...An, all branch registers B1...Bm, the last choice point LCP, the environment register CE, the mode register M, are saved in the choice point.

if_false_jump Vn, Code This instruction appears after the condition in an if-then-else in the body. If the value in variable **Vn** is not **true**, the program pointer P is set to Code.

jump Code This instruction appears after the then clause of an if-then-else in the body. This instruction sets the program pointer `P` unconditionally to `Code`.

4.8.2 Get instructions

The *get* instructions are used to match the arguments of a call against the head of a rule. They are different from the WAM *get* instructions because Prolog does unification rather than matching, and hence the arguments of a call in Prolog could have unbound variables. In SEL, checking if the value in the register is an unbound variable is unnecessary because all SEL arguments must be ground terms.

get_variable Vn, Ai This instruction appears if the term in the head of the rule is a variable and it is the first occurrence of that variable. The instruction simply assigns the value in register `Ai` to variable `Vn`.

get_value Vn, Ai This instruction appears if the term in the head of the rule is a variable and it is not the first occurrence of that variable in the rule. The instruction checks if the values in register `Ai` and variable `Vn` match. If it does not, it sets the *fail* flag.

get_nil Ai This instruction appears if the term in the head is `[]`. The instruction checks if register `Ai` holds the null list. If it does not then it sets the *fail* flag.

get_phi Ai This instruction appears if the term in the head is {}. The instruction checks if register `Ai` holds the null set. If it does not, it sets the *fail* flag.

get_const C, Ai This instruction appears if the term in the head is an integer, atom or boolean. The instruction checks if register `Ai` has the value `C`. If it does not, it sets the *fail* flag.

get_list Ai This instruction is used when a list-pattern appears in the head of a rule. It checks if register `Ai` is pointing to a list. If so, it assigns the structure pointer `S` to it. If it does not, the *fail* flag is raised.

get_set Ai, Vn This instruction followed by an "adjust" instruction are used when a set-pattern appears in the head of a rule. The instruction checks if register `Ai` holds a set. If it does, the variable `Vn` gets the value of `Ai`; otherwise the *fail* flag is set.

adj_set_head Vn This instruction appears after a get-set instruction if there is a set-pattern with the don't-care variable "_" following the "|" in the set pattern. This "_" variable indices that the remainder of the set does not have to be constructed. The structure pointer S gets the value of `Vn`, and `Vn` is adjusted to point to the remainder of the set. The current branch pointer `CB` is incremented by one and the branch register `B(CB)` (the `CB`-th branch register) is made to point to this instruction.

adj_set Vn This instruction appears after a get_set instruction if the remainder is a variable and the remainder does not have to be copied. The structure pointer gets the value of `Vn` and `Vn` is adjusted to point to the next element of the set as head and tail. The current branch pointer is incremented by one and the branch register `B(CB)` is made to point this instruction.

adj_set_with_copy Vn This instruction is just like the adj_set except that the tail of the set is copied each time the set is adjusted.

4.8.3 Put Instructions

The *put* instructions are used to load the registers with the arguments of a call. These instructions are similar to the *put* instructions of the WAM.

put_nil Ai* This instruction appears if the argument to a call is the null list. It loads the register `Ai` with `[]`.

put_phi Ai This instruction appears if the argument to a call is the null set. It loads the register `Ai` with {}.

put_const C,Ai* This instruction appears if the argument to a call is a integer, atom or boolean. It loads the register `Ai` with the constant C.

put_variable Yn, Ai* This instruction is used only to load the location for the result of a function call. A pointer to `Yn` (address of `Yn`) is stored in register `Ai`.

put_value Vn, Ai* This instruction is used for the second or later occurrence of a variable as an argument to a function call. The instruction assigns the value in variable `Vn` to register `Ai`.

put_list Ai* This instruction is used if the argument to a function call is a list. The instruction puts a list pointer to the top of heap in `Ai`.

put_set Ai This instruction is used if the argument to a function call is a set. The instruction puts a set pointer to the top of heap in `Ai`.

4.8.4 Store Indirect Instructions

The store indirect instructions are used to return values. If we view the location where the answer is returned as an extra argument of the corresponding Prolog predicate, these instructions are the WAM's *get* instructions. The difference is that here we know at compile time when the register is going to have an unbound variable.

store_ind_nil Ai This instruction is used if the right hand side of a rule is the null list. It sets the variable pointed to by `Ai` equal to `[]`.

store_ind_phi Ai This instruction is used if the right hand side of a rule is the null set. It assigns the null set to the variable pointed to by `Ai`.

store_ind_const C,Ai This instruction is used if the right hand side of a rule is an integer, atom or boolean. It assigns the constant C to the variable pointed to by `Ai`.

store_ind_list Ai This instruction is used if the right hand side is a list. It sets the variable pointed to by `Ai` to a list pointer pointing to top of heap.

store_ind_set Ai This instruction is used if the right hand side is a set. It sets the variable pointed to by `Ai` to a set pointer pointing to top of heap.

store_ind_value Vn, Ai This instruction is used if the right hand side of a rule is a variable. The instruction makes the value of the variable pointed to by `Ai` equal to that of the value in `Vn`.

store_ind_variable Vn,Ai This instruction is used if the right hand side of a rule is a function call or if it is a subset rule. The instruction stores the reference to the unbound variable in `Ai`, in `Vn`.

4.8.5 Match and Store Instructions

These instructions are used to match nested patterns and to load nested arguments. These are the WAM's *unify* instructions in *read* and *write* mode. Note that there is no need for an instruction similar to the WAM's unify_local_value.

match_variable Vn This instruction is used if there is a variable within a list or set pattern in the head and it is the first occurrence of that variable in the rule. The instruction gets the value pointed to by the structure pointer `S`, and stores it in `Vn`. `S` is incremented by 1.

match_value Vn This instruction is used if there is a variable within a list or set pattern in the head and it is not the first occurrence of that variable in the rule. The instruction checks if the value pointed to by the structure pointer matches the value in `Vn`. If it does not, the *fail* flag is set. `S` is incremented by 1.

match_nil This instruction is used if `[]` occurs within a list or set pattern in the head of a rule. If the value pointed to by the structure pointer `S` is not equal to `[]`, then the *fail* flag is set.

match_phi This instruction is used if {} occurs within a list or set pattern in the head of a rule. If the value pointed to by the structure pointer `S` is not equal to the null set, then the *fail* flag is raised.

match_const C This instruction is used if there is an integer, boolean or atom in alist or set pattern in the head of a rule. If the value pointed to by the structure pointer `S` is not equal to `C`, the *fail* flag is assigned true.

store_nil This instruction is used if the null list is part of the structure that is an argument to a function call. The null list is pushed on top of heap.

store_phi This instruction is used if {} is part of the structure that is an argument to a function call. The null set is pushed on top of heap.

store_const C This instruction is used if there is an integer, boolean or atom as part of a structure that is an argument to a function call. The constant `C` is pushed on to the heap.

store_variable Vn This instruction is used if there is a function call as part of a structure that is an argument to another function call. A new unbound variable is pushed on top of heap and a reference to it is stored in `Vn`.

store_value Vn This instruction is used if there is a variable as part of a structure that is an argument to a function call. The value in `Vn` is pushed on top of heap.

4.8.6 Indexing Instructions

There are fewer indexing-instructions here than in the WAM.

try_equ_else Proc This instruction precedes every equality rule that has at least one more rule with the same first argument in the rule head. `Proc` is the address of the following rule. All registers are stored in an alternate register set to try the other rule if this fails to match. The alternate program counter is set to `Proc`.

try_sub_and Proc This instruction precedes every subset rule that has at least one more subset rule with the same kind of first head argument. A choice point is created and all registers `A1...An`, the last choice point `LCP`, the current environment pointer `CE` and the continuation pointer `CP` etc are saved in the choice point.

switch_on_ground_term Lc.Ll,Ls This instruction is used to switch control to different groups of clauses dependent on the first argument of the function call. The program pointer `P` is set to `Lc`, `Ll` or `Ls` depending on whether the register `A1` holds a constant, list or set.

4.9 Comparison with Prolog

In the following table, we compare our implementation with a couple of Prolog implementations. The table gives the time taken in milliseconds (on a Sun 3/60) for a few programs run under the C-prolog interpreter, the Quintus Prolog compiler and the SEL interpreter. The actual Prolog programs used are given in appendix C. Note that when "setof" operations are used the SEL interpreter is faster than

C-Prolog, but slower than Quintus Prolog. This is because of the highly optimised code generated and runtime emulation by Quintus Prolog.

goal	C-Prolog	Quintus Prolog	SEL
reverse of a 30 element list	166	33	266
prod of 2 20-element sets	250	84	416
perms of a 6 element sets	16350	2200	5866
all solutions to the 8-queens	173633	33766	75000
subset testing of two 20 element sets	150	17	66

5 Conclusions

This thesis has presented techniques similar to the WAM [W83] for compiling subset logic programs. We have described instructions for the compilation of restricted a-c matching and the compilation of the depth-first control strategy. The multiple a-c matches are represented by multiple "branch-points" within WAM's choice point. A distinct feature of the control is that we have backtracking on *failure* (as in Prolog) as well as on *success* (to collect all elements of a set).

The property of functions "distributing over union" was made use of in the implementation by providing two modes of calling a function: `call-one` and `call-all`. It was found that this property does aid in reducing execution time (because it obviates duplicate checking), but the extent of improvement is usually diminished by the increase in the number of function calls to be performed. Explicitly annotating functions to bypass duplicate checking has also been found very useful.

In the process of undertaking this implementation, we found that the basic computational model for subset assertions could be readily adapted to support quantifiers over sets (the `ifall` and `ifone` constructs). From our experience using them, we found that they lead to short programs that are also efficient.

This thesis has concentrated on run-time issues rather than compile-time issues. There appears to be many opportunities for global compile time analysis. For example, the compiler should be able to check in many cases the confluence of equality assertions and also the property of distribution over union. It is also desirable to have methods that could tell us which definitions could possibly generate duplicate elements in a set and do duplicate checking only in those cases. These are probably undecidable issues, but partial (correct) information could be provided which is better than no information. Global analysis is also required to determine when a-c matching may destructively "adjust a set" and when it has to make separate copies of remainder sets. Type inference is another area to be investigated which could lead to substantial savings in run time error checking of operations such as union of sets and arithmetic.

The current implementation does not perform garbage collection of the heap, which would be desirable in an improved version. Runtime extensions are also needed to support set closures [JP88].

A UNIX Man Page

```
SEL(1) UNIX Programmer's Manual SEL(1)

NAME
sel - interpreter for SEL

SYNOPSIS
sel [ option ]

DESCRIPTION
SEL is a subset logic language designed for
programming with sets. It uses equality and
subset assertions to define functions.

The system (SEL Version 1.0) consists of a
compiler that compiles SEL programs into an
instruction set for an abstract machine, and
an emulator for the instruction set. It is
an integrated package to experiment with
SEL programs. It features some elementary
trace facilities and integer arithmetic.

The following options are recognized.

-a Prints the assembled instructions
of the abstract machine for each
rule and goal
-I Prints runtime statistics at the
instruction level
-C Generates call-all at the outermost
level
```

AUTHORS

The system was written by Anil Nair using
lex, yacc and C.

SEE ALSO

Subset Logic Programming: Application and
Implementation
B.Jayaraman and Anil Nair, In Fifth
Int'l Logic Programming Conference,
pp. 843-858, Seattle, 1988.

Compilation of Subset Logic Programs
Anil Nair (M.S. Thesis), UNC, Chapel Hill,1988.

BUGS

Report to bj@cs.unc.edu.

B A Sample Session

```
unc % sel
SEL Version 1.0

sel> compile('queens').
[iota, queens, solve, placequeen, safe, ]
sel> cputime().
66
sel> solve(4).
{{[4|3],[3|1],[2|4],[1|2]},{[4|2],[3|4],[2|1],[1|3]}}
sel> cputime().
250
sel> trace(queens).
sel> solve(4).
Call to queens(4,1,{},{4,3,2,1}).
Call to queens(4,2,{[1|4]},{4,3,2,1}).
Call to queens(4,3,{[2|2],[1|4]},{4,3,2,1}).
Call to queens(4,3,{[2|1],[1|4]},{4,3,2,1}).
Call to queens(4,4,{[3|3],[2|1],[1|4]},{4,3,2,1}).
Call to queens(4,2,{[1|3]},{4,3,2,1}).
Call to queens(4,3,{[2|1],[1|3]},{4,3,2,1}).
Call to queens(4,4,{[3|4],[2|1],[1|3]},{4,3,2,1}).
Call to queens(4,5,{[4|2],[3|4],[2|1],[1|3]},{4,3,2,1}).
Call to queens(4,2,{[1|2]},{4,3,2,1}).
Call to queens(4,3,{[2|4],[1|2]},{4,3,2,1}).
Call to queens(4,4,{[3|1],[2|4],[1|2]},{4,3,2,1}).
Call to queens(4,5,{[4|3],[3|1],[2|4],[1|2]},{4,3,2,1}).
Call to queens(4,2,{[1|1]},{4,3,2,1}).
Call to queens(4,3,{[2|4],[1|1]},{4,3,2,1}).
Call to queens(4,4,{[3|2],[2|4],[1|1]},{4,3,2,1}).
Call to queens(4,3,{[2|3],[1|1]},{4,3,2,1}).
{{[4|3],[3|1],[2|4],[1|2]},{[4|2],[3|4],[2|1],[1|3]}}
```

```
sel> ^D
SEL execution halted
unc %
```

C Prolog Programs

This appendix gives the listings of the Prolog programs used in section 4.9.

```
rev([], []).
rev([H|T], Z) :- rev(T, Y), append(Y, [H], Z).

append([], X, X).
append([H|T], Y, [H|Z]) :- app(T, Y, Z).

prod([],Y,[]).
prod([X|Y],S,T) :- distr(X,S,U),prod(Y,S,V),append(U,V,T).

distr(X,[],[]).
distr(X,[H|T],[[X|H]|Z]) :- distr(X,T,Z).

perm([], []).
perm(L, [E|X]) :- select(E, R, L), perm(R, X).

select(X, L, [X|L]).
select(Y, [X|L2], [X|L]) :- select(Y, L2, L).

allperms(L, P) :- setof(X, perm(L, X), P).

solve(Board_size, All_Soln) :-
bagof(Soln,queens(Board_size, [], Soln),All_Soln).

%       queens accumulates the positions of occupied squares

queens(Bs, [square(Bs, Y) | L], [square(Bs, Y) | L]) :- size(Bs).
queens(Board_size, Initial, Final) :-
        place(Initial, Next),
        queens(Board_size, [Next | Initial], Final).
```

```
mem(H,[H|_],t).
mem(H,[X|Y],t) :- mem(H,Y,t).

subset([],X,t).
subset([H|T],S,t) :- mem(H,S,t),subset(T,S,t).
subset(X,Y,f).
```

```
%       place generates legal positions for next queen

place([], square(1, X)) :- snint(X).
place([square(I, J) | Rest], square(X, Y)) :-
        X is I + 1,
        snint(Y),
        not(threatened(I, J, X, Y)),
        safe(X, Y, Rest).

not(G) :- G,!,fail.
not(G).

%       safe checks whether square(X, Y) is threatened by any
%       existing queens

safe(X, Y, []).
safe(X, Y, [square(I, J) | L]) :-
        not(threatened(I, J, X, Y)),
        safe(X, Y, L).

%       threatened checks whether squares (I, J) and (X, Y)
%       threaten each other

threatened(I, J, X, Y) :- (I = X), !.
threatened(I, J, X, Y) :- (J = Y), !.
threatened(I, J, X, Y) :- (U is I - J), (V is X - Y), (U = V), !.
threatened(I, J, X, Y) :- (U is I + J), (V is X + Y), (U = V), !.

snint(1).  snint(2).  snint(3).  snint(4).
snint(5).  snint(6).  snint(7).  snint(8).

size(8).
```

References

[CM81] W.F. Clocksin and C.S. Mellish, " Programming in Prolog," Springer-Verlag, New York, 1981.

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

[JP88] B. Jayaraman and D.A. Plaisted, "Semantics of Subset-logic Programming," Manuscript, July 1988.

[JN88] B. Jayaraman and Anil Nair, "Subset-Logic Programming: Application and Implementation", In *Fifth Int'l Logic Programming Conference*, pp.843-858 , Seattle, 1988.

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

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

[W83] D. H. D. Warren, "An Abstract Prolog Instruction Set,", Technical Report 309, SRI International. October 1983.