Implementation of Subset Logic Languages

Ph.D. Dissertation Defense

Kyonghee Moon

Department of Computer Science
State University of New York at Buffalo

Outline

1. Contributions
2. Background and Related Work
3. The SuRE Programming Language: Overview
4. Abstract Machine for SuRE
5. Static Analysis
6. Experimental Results
7. Further Work

1. Contributions of this Dissertation

The design and implementation of a declarative language with powerful set processing capabilities

- Use of sets for unifying functional and logic programming
- Novel use of memo-tables and lazy-evaluation
  - Monotonic Memo-table
  - Lazy exploration of Search Space
- Static Analysis for subset clauses
  - Distribution over Union
  - Detection of Memoization and Avoidance of Re-do

2. Background and Related Works

- Sets in Programming Languages
  - imperative language (Pascal, SETL)
  - Functional Language (Miranda)
  - Declarative Language (Prolog)
- Warren Abstract Machine (WAM)
  - Abstract Machine for Prolog
  - Compilation of unification and control stragegy into high-level instructions
  - Run-time stack (environments and choice points) and Heap
- XSB & CORAL
  - XSB:
    - SLG resolution, a table-oriented resolution that combines SLD resolution with memoization
    - does not support set terms
  - CORAL:
    - deductive database system that employs bottom-up evaluation
    - supports ground sets and multisets

Warren Abstract Machine (WAM)

Unification : get, unify, put instruction.

e.g. `eq(X, X).`

```
get_variable X2, A1
get_value    X2, A2
```

e.g. `gp(X, Y) :- p(X, Z), p(Z, Y).`

```
get_variable Y1, A1
get_variable Y2, A2
put_value Y1, A1
put_variable Y3, A2
call p/1
put_value Y3, A1
put_value Y2, A2
execute p/1
```

Control Strategy: Indexing Instructions

`switch_on_term, try_me_else, retry_me_else, trust_me.`

- link together the different clauses that make up a procedure and are responsible for filtering out a subset of those clauses that could potentially match a given procedure clause and responsible for backtracking.

Example

```
append([], L, L).
append[X|L1], L2, [X|L3]) :- append(L1,L2,L3).

append/3: switch_on_term C1a,C1,C2,fail

C1a: try_me_else C2a
C1:  get_nil A1
     get_variable X3, A1
     get_value X3, A2
     proceed

C2a: trust_me_else fail
C2:  get_list A1
     unify_variable X4
     unify_variable X5
     get_variable X6, A2
     get_list A3
     unify_value X4
     unify_variable X7
     put_value X5, A1
     put_value X6, A2
     put_value X7, A3
     execute append/3
```

SEL = **S**ubset-**E**quational **L**anguage:

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

SEL is a proper subset of **SuRE**, and is a functional programming language.

- Compilation of set-matching
- Specification of functions that distribute over union via mode declaration
- No detection of cyclic calls - No memoization

SuRE = SEL + SRL + conditional equational clause

3. The SuRE Language: Overview

SuRE = **Su**bset, **R**elational, and **E**quational Language

Equational Clause:

f(terms) `equals` *expression.*

- many Prolog programs are essentially functional programs [DW89]
- equations are clearer (no cuts) and more efficient (no backtracking)

Subset Clause:

f(terms) `contains` *expression.*

- helps avoid common uses of *setof* and *modes*
- helps avoid some uses of *assert*, *retract*, and *cut*
- efficient formulations of set operations, including transitive closures and dynamic programming algorithms
- helps render clear and concise formation to problems involving aggregate operations and recursion in database query.

3.1 Syntax of SuRE Programs

3.2 Set Constructors in SuRE

3.3 Stratified SuRE Programs

3.4 From Subset to Partial-order Clauses

3.5 Lazy Enumeration

3.1 Syntax of SuRE Programs

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

f(terms) `equals` *expr.*
f(terms) `equals` *expr* `:-` *condition.*

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

where
term is made of constants, variables, constructors, `{_\_}`,

term' is same as *term* except it uses a set term of `{_/_}`,

expr is made of terms and user-defined function symbol, and

condition is made of $p(terms)$, *not* $p(terms)$, and *set enumeration goals*:

(i) $f(terms) \ni term$: incremental enumeration of set elements

(ii) `lazy` $f(terms) = set$: lazy enumeration of set elements

(iii) $f(terms) = set$: eager enumeration of set elements

3.2 Set Constructors in SuRE

Sets are represented by two novel constructors: $\{x\backslash t\}$ & $\{x/t\}$

phi stand for the empty set

$\{x\backslash t\}$ stands for a set s where $x \in s$ and $t = s - \{x\}$.

e.g. Matching $\{x\backslash t\}$ against $\{1, 2, 3\}$ yields:

$x \leftarrow 1, t \leftarrow \{2, 3\}$

$x \leftarrow 2, t \leftarrow \{1, 3\}$

$x \leftarrow 3, t \leftarrow \{1, 2\}$

$\{x/t\}$ stands for a set $s = \{x\} \cup t$

	l.h.s. of fun	*l.h.s. of pred*	*r.h.s. of fun*	*r.h.s. of pred*
$\{x\backslash t\}$	$\surd$	X	X	X
$\{x/t\}$	X	$\surd$	$\surd$	$\surd$

Set Terms in Relations and Set-of Operations

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

- to verify set membership, `?- member(b, {a,b,c}).`

- to generate the elements of a set one at a time, `?- member(X, {a,b,c})`

- to insert an element in a set, `?- member(a, S).`

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

The Prolog goal

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

can be replaced using a conditional subset clause, as follows,

```
parts(List) contains {[X, Y]} :- append(X, Y, List).

?- parts([1,2,3]).
```

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

Subset clauses often yield compact, nonrecursive definitions:

```
union(S1,S2) contains S1
union(S1,S2) contains S2

setproduct({X\_}, {Y\_}) contains {[X|Y]}

intersect({X\_}, {X\_}) contains {X}

? setproduct({1,2}, {3,4})

{[1|3],[1|4],[2|3],[2|4]}
```

3.3 Stratified SuRE Programs

Informal Definitions

- A logic program is said to be *stratified* if there are no recursive calls on a predicate through negation.

- A logic program is said to be *locally stratified* if there are no *cyclic* calls on a predicate through negation

- A function p is subset-monotonic in some argument if

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

• Recursive Subset Clause

- Commonly arises in recursive subset clauses of the form

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

Example 1:

list permutations of a set

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

distr(X, {H\_}) contains {[X|H]}.
```

• Memoization for loop detection

Example 2:

```
memo reach/1/phi.

reach(V) contains {V}.
reach({X\_}) contains reach(edge(X)).

edge(1) contains {2}.
edge(2) contains {1}.
```

In general, they are useful in problems such as data-flow analysis in compilers, graph theory, etc.

A **memo-table** is a (run-time) data structure to record the result of a function call, so subsequent calls of that function with identical argument can be reduced to table look-up.

There is more to a memo function than just memoization ...

• Cyclic function definition via monotonic function

Example 3: Dataflow Analysis in a Compiler [AU77]

```
memo out/1, in/1, allout/1.

out(B) contains diff(in(B), kill(B)).
out(B) contains gen(B).

in(B) contains allout(pred(B)).

allout({P\_}) contains out(P).

diff({X\_},S) contains if member(X,S) then phi else {X}.
```

When general circular containment constraints exist, there is a need to assume a provisional value for the inner recursive call (when the loop is detected), and to revise this estimate and re-execute the call until a *fixed point* is reached. This process is guaranteed to yield a unique answer if one memo function is defined in terms of another through *subset-monotonic* functions.

The Need for Re-do

We explain how re-do works for a smaller example ...

3.4 From Subset to Partial-order Clauses

Examples of Monotonic Aggregation

Company-controls

```
memo controls/2/false.

controls(X,Y) equals sum(owns(X,Y)) > 50.

owns(X,Y) contains {s(X,Y,N)} :- shares(X,Y,N).
owns(X,Y) contains {s(Z,Y,N)} :- shares(Z,Y,N), control

sum(phi) equals 0.
sum({s(_,_,C)\T}) equals C+sum(T).
```

Shortest-path

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

3.5 Lazy Enumeration

Lazy goal, *lazy* $f(\bar{t}) = s$, means s is the set of values generated by $f(\bar{t})$, but the elements of s are generated lazily.

Example:

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

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

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

Call-one vs. Lazy call

- Call-one invocation of a subset goal simply generates elements one at a time

- Lazy call accumulates all generated elements as needed

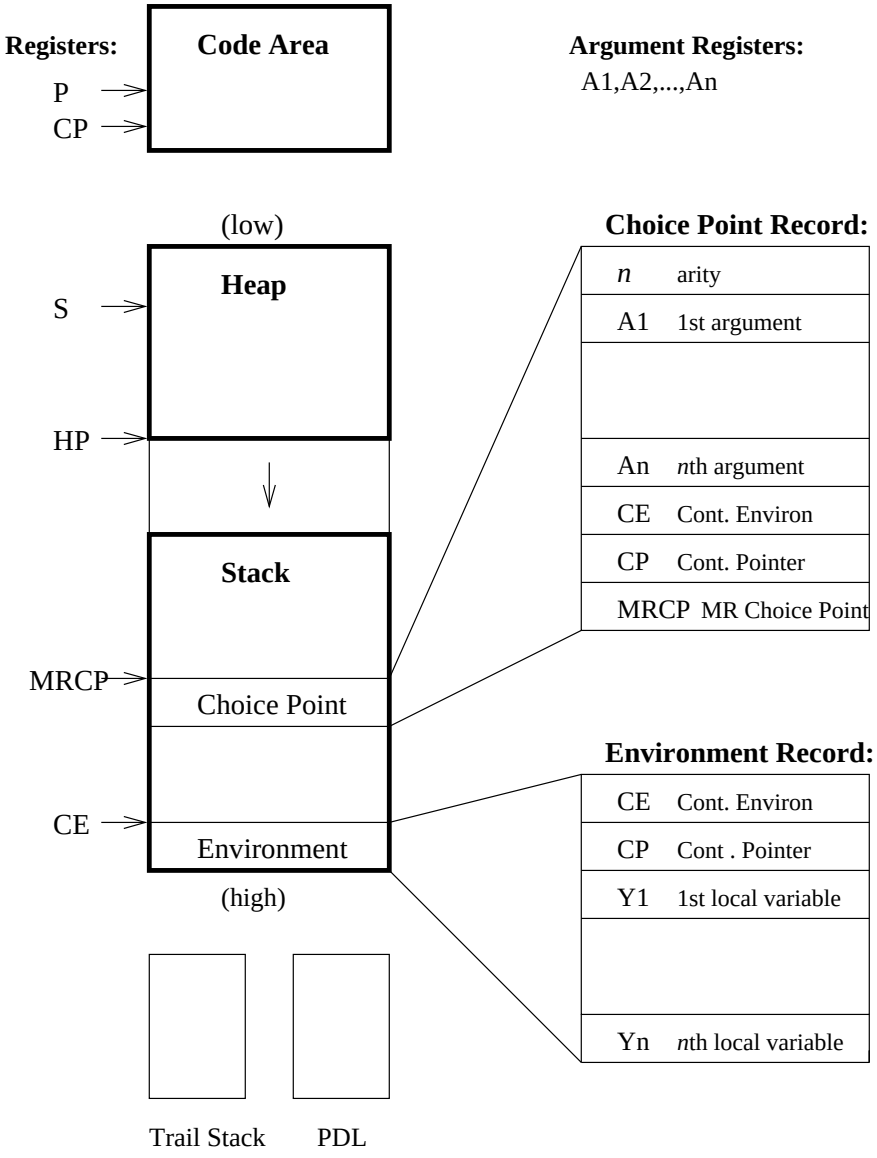
Approach to implementation of lazy call

- Uses partially constructed set with *read-only* variable as the remainder set

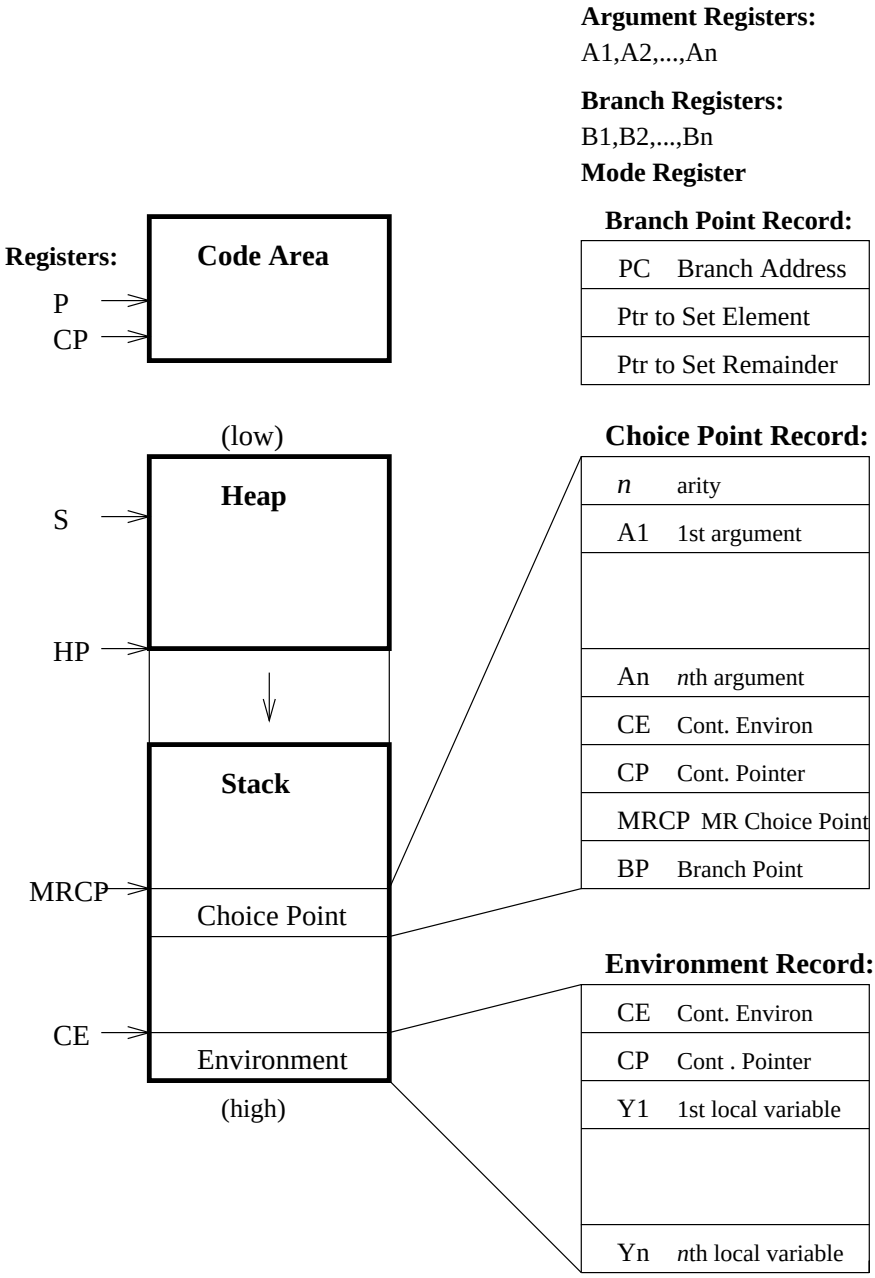
- Each such read-only variable is unique and has a resumption point associated with it, where further elements can be obtained

4. Abstract Machine for SuRE

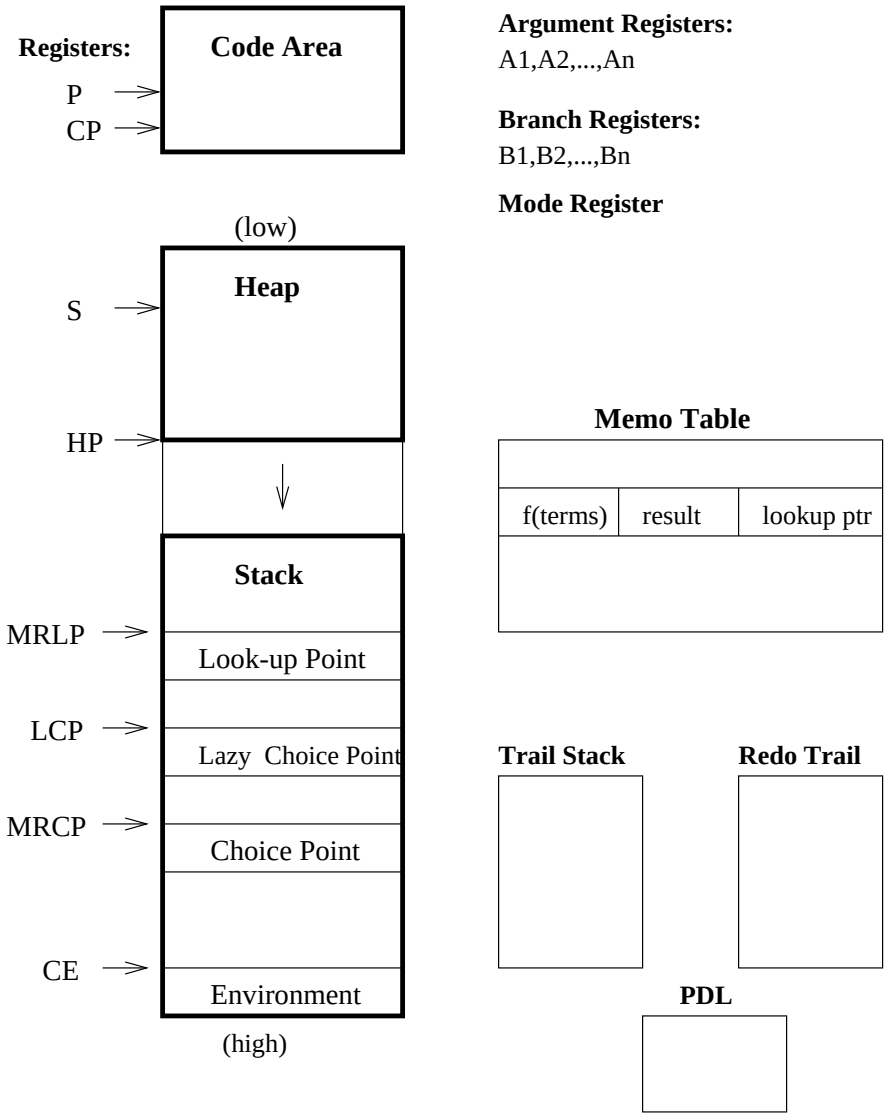
Warren Abstract Machine



Abstract Machine for SEL



Extended WAM for SuRE



Comparing Execution Models of SEL, SuRE, and Prolog

	Data	Control
Prolog	Unification	Backtracking
SEL	Set-Matching	Collect-all Call-one/Call-all Branching Memoization Re-Do
SuRE	Set-Unification	Lazy Set Enumeration

Instruction Sets

Get Instructions

get_variable V_n, A_i get_value V_n, A_i
get_constant C, A_i get_nil A_i
get_structure F, A_i get_list A_i
get_set A_i, V_j

Match Instructions

match_variable V_n match_variable V_n
match_value V_n match_value X_n
match_constant C match_nil

Set-Matching Instructions

adj_set_head Y_i
adj_set Y_i
adj_set_with_copy Y_i

Store_indirect Instructions

store_indirect_variable V_n, A_i store_indirect_value V_n, A_i
store_indirect_set A_i store_indirect_list A_i
store_indirect_str n, A_i store_indirect_const C, A_i
store_indirect_nil A_i store_indirect_phi A_i

Put Instructions

put_variable V_n, A_i put_value V_n, A_i
put_constant C, A_i put_nil A_i
put_structure F, A_i put_list A_i
put_set A_i

Store Instructions

store_variable V_n store_variable V_n
store_value V_n store_value X_n
store_constant C store_nil

Unify Instructions

unify_variable V_n unify_variable V_n
unify_value V_n unify_value X_n
unify_constant C unify_nil

Set-Unification Instructions

w_getstructure F, Y_i w_getlist Y_i
w_getset Y_i

write_variable V_n write_variable V_n
write_value V_n write_value X_n

write_constant C write_nil
write_phi

Procedural Instruction

call_all P, N call_one P, N
last_call_one P execute P
call_memo P, N execute_memo P
call_lazy P, N execute_lazy P
execute_nr P, N
allocate deallocate
collect V_i, V_j
updatememo V_i, V_j
proceed

Indexing Instruction

switch_on_ground_terms switch_on_terms
try_equ_else try_me_else
try_sub_and trust_me

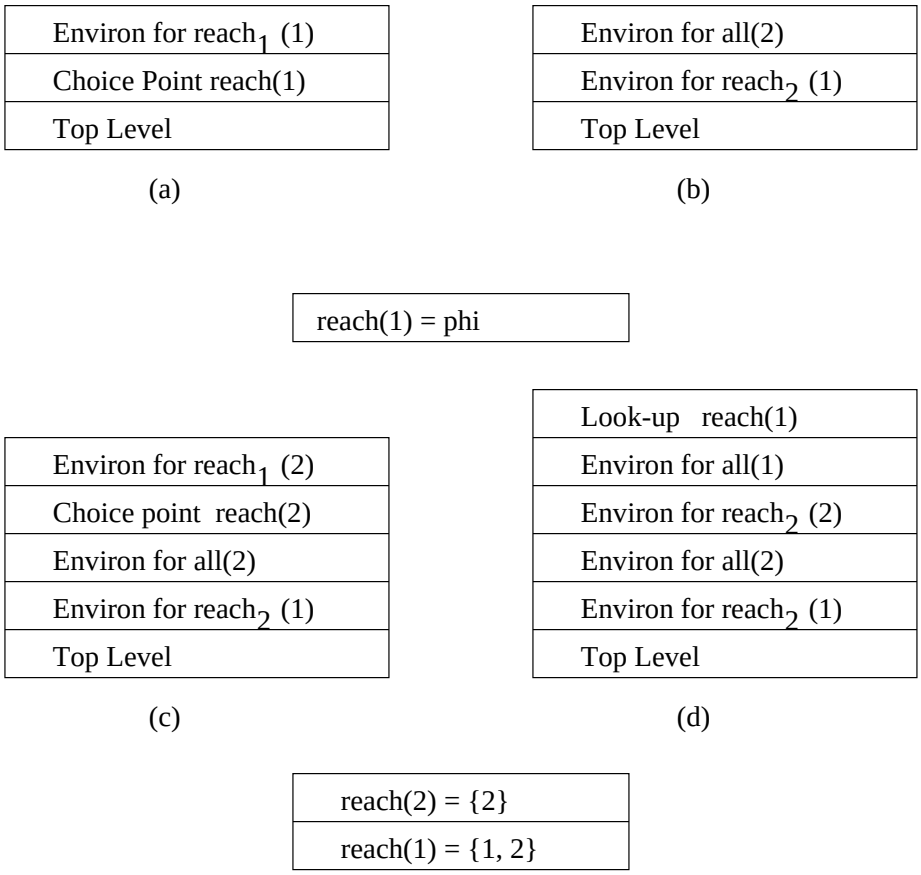
```
      reach(V) contains {V}.
      reach(V) contains reach(W) :- edge(V, W).

      edge(1, 2). edge(2, 1).

      reach(V) contains S1 :- edge(V, W), all(W) = s1.
      all(W) contains reach(W).

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        % all(W)
         putvariable     Y4, A2        % = S2
```

```
        execute         all/1
        collect         Y2, Y4          % S1 := S1 ∪ S2
all:    label           all/1
        allocate        3
        getvariable     Y1, A1          % all(W)
        save_choice_point               % contains
        storeindvar     Y2, A2          % S1 :-
        putvalue        Y1, A1          % reach(W)
        putvariable     Y3, A2          % = S2
        execute_memo    reach/1
        update_memo     reach, 1, Y3
        collect         Y2, Y3          % S1 := S1 ∪ S2
```

4.1 Implementation of Memo Functions

4.2 Set Unification

4.3 Implementation of Lazy Enumeration


Environ for reach$_1$ (1)
Choice Point reach(1)
Top Level
(a)
Environ for all(2)
Environ for reach$_2$ (1)
Top Level
(b)
reach(1) = phi
Environ for reach$_1$ (2)
Choice point reach(2)
Environ for all(2)
Environ for reach$_2$ (1)
Top Level
(c)
Look-up reach(1)
Environ for all(1)
Environ for reach$_2$ (2)
Environ for all(2)
Environ for reach$_2$ (1)
Top Level
(d)
reach(2) = {2}
reach(1) = {1, 2}

4.1 Implementation of Memo Functions

Instructions and Its Operations

`call_memo` and `execute_memo` f/n

```
if f/n is called first time
   insert into memo-table
else
   look-up the memo-table
```

Insert into Memo-table

- Create the memo-table entry for $f(\overline{t})$ and initialize its value to *phi* or an initial value given.

Look-up

- Retrieve the value of $f(\overline{t})$ and control transfers to the next instruction instead of executing $f(\overline{t})$

- Create *look-up point frame* on the stack and save necessary information for re-do

Updating memo-table

- When the computation of $f(\overline{t})$ is completed

- If the new value of $f(\overline{t})$ is different from old value and there is an intervening look-up, a re-do is triggered

Re-Do

- Control is transferred to the point where the look-up of $f(\overline{t})$ occurred for the first time

4.2 Set Unification

Rules [JP89, JJ94]:

1. $\{t_1/s_1\} = \{t_2/s_2\}$
   (a) $t_1 = t_2$, $s_1 = s_2$
   (b) $s_1 = \{t_2/z\}$, $s_2 = \{t_2/z\}$ where z is a new variable
2. $x = \{t_1/s_1\}$
   (a) x does not occur in $\{t_1/s_1\}$: $x \leftarrow \{t_1/s_1\}$.
   (b) x occurs in t_1. The equality is unsolvable.
   (c) x does not occur in t_1 but occurs in s_1. The equality is solvable iff s_1 is either x or $\{t_2/x\}$ or $\{t_2/\{t_3/x\}\}$, etc. where x does not occur in any t_i. The unifying substitution is respectively either $x \leftarrow \{t_1/z\}$, or $x \leftarrow \{t_1/\{t_2/z\}\}$, etc., where z is a new variable.
3. $\{x_1, \ldots, x_m/s\} = \{y_1, \ldots, y_n/s\}$ and for $i = 1, \ldots, m, j = 1, \ldots, n$, $x_i \neq y_j$: $s \leftarrow \{x_1 \ldots x_m, y_1, \ldots, y_n/z\}$, where z is a new variable.
4. $\{x_1, \ldots, x_m/s\} = \{y_1, \ldots, y_n/s\}$ and some of the x_i's and y_i's are identical. Generate a new equation by deleting common ground terms on both sides of the equality.
5. $\{x/s\} = \{x/\{x/s\}\}$

4.3 Implementation of Lazy Evaluation

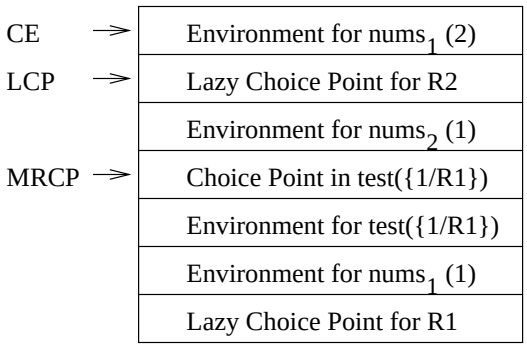
- A new read-only variable will be generated only when the current calling mode is lazy and there is an outstanding choice points related to the current environement. Such choice point is designated as a lazy choice point.

- Whenever a read-only variable is matched against a set pattern $\{x \backslash t\}$ or with another read-only variable, the next element of the corresponding set is generated with a new read-only variable attached.

- To protect the search space assocated with a read-only variable from normal backtracking, MRCP is updated to point to the most recent choice point below the environment of the lazy call.

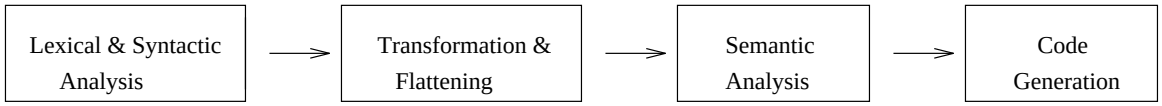
Example

```
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).
```



Compiler and Run-Time System

The phases of compilation are as follows:



- Compiler 12900
  - Syntactic Analysis 2700
  - Static Analysis 2800
- Emulator 15200
  - Memoization 870
  - Lazy Enumeration 400

5. Static Analysis

• Static Analysis is a form of compile time analysis to gather information that will assist in producing a more efficient program.

• Static Analysis in Imperative Language

e.g. available expressions analysis for common subexpression elimination, reaching definitions analysis for code motion.

• Static Analysis in Prolog

- Efficient Compilation of Unification
- Mode Inference

• Static Analysis in SuRE

- Correctness
  * Groundness Analysis
- Efficiency
  * Detection of Distribution-over-Union
  * Detection of Memo function and Avoidance of Re-Do

5.1 Distribution Over Union

An operation f *distributes over union* in some argument if

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

e.g. `union`, `setproduct`, `intersect` distribute over union in both their argument positions. However, the following function, `perms`, does not distribute over union in its argument (but the function `distr` does in its second argument):

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

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

Advantages

- can avoid checking for duplicate elements

- can avoid forming intermediate sets

Potential Disadvantage

- might overcompute if there are duplicates

Call-one mode vs. Call-all mode

call-one mode : when a set-valued function is called to produce one element of its resulting set rather than the entire set.

call-all mode : when a set-valued function is called to return the entire set.

Approach

1. A set constructor `{H\T}` appears in an argument position at the head of the clause, and the remainder set `T` is not subsequently used

2. A variable appears in an argument position 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 arugment position.

Example:

```
scc({X\T},E) equals scc1(int(reach(X,E),b\_reach(X,E)), {X\T

scc1(S,_,_) contains {S}.
scc1(S,T,E) contains scc(diff(T,S),E).

reach(X,_) contains {X}.
reach(X,E) contains allreach(dadj(X,E),E).

allreach({X\_},E) contains reach(X,E).

dadj(X,{[X|Y]\_}) contains {Y}.

b_reach(X,_) contains {X}.
b_reach(X,E) contains b_allreach(b_adj(X,E),E).

b_allreach({X\_},E) contains b_reach(X,E).
```

```
b_adj(X,{[Y|X]\_}) contains {Y}.

% Helper Functions

int({X\_},{X\_}) contains {X}.

diff({X\_}, S) contains if member(X,S) then phi else {X}.

member(X,{X\_}) equals true.
member(_, _)    equals false.
```

Limitation

In general, if a cycle is involved in the call sequence, the algorithm will fails to detect the property.

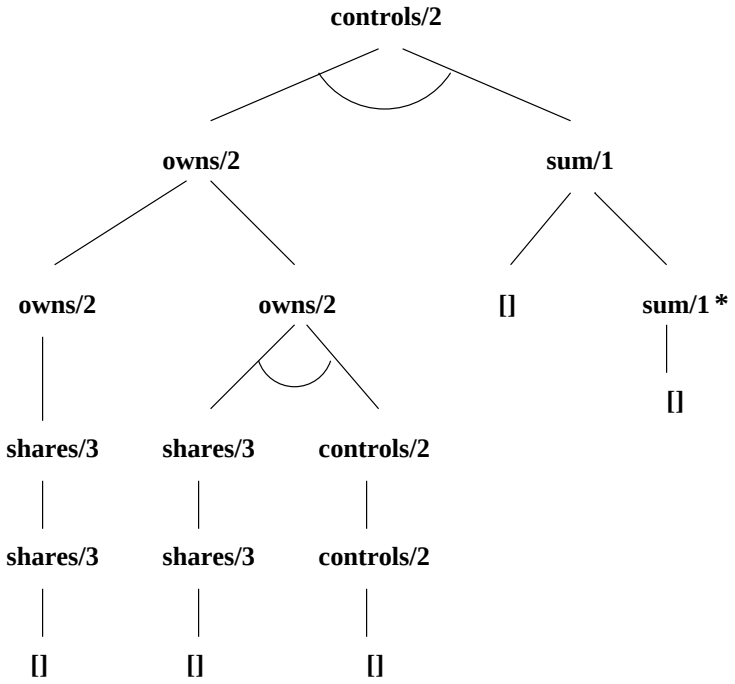
5.2 Memoization and Re-Do

Detection of Memo functions

- Builds an AND-OR tree of or a forest of AND-OR tree

- A cycle can be detected easily by doing a preorder traversal on it.

Example:



Detection of No Re-Do

- When there is no function that takes the value of closure function as an argument, there is no need for a re-do

- Such case can be detected syntactically during flattening stage of compilation and can be proven to be sound [OJ93]

5.3 Groundness Analysis

Correctness

groundness of negated goal: to ensure soundness of negation-as-failure

groundness of subset and equational goal: argument to subset and equational clause must be ground

Efficiency

efficient compilation of set terms in relational goal: set unification can be reduced to set matching

Approach

SuRE = SEL + SRL

subset clauses in SEL : $f(terms)$ `contains` $expr$

subset clauses in SRL : $f(terms)$ `contains` $expr$ `:-` $Cond$

- calling a function defined in SEL from SRL : the argument to a function must be certified to be ground

- calling a function defined in SRL from SEL : the argument to a function is guaranteed to be ground

Analysis Phases

1. Identify variables that must be ground because they are used in functional goal in the clauses body - initial calling pattern of a relational clause

2. Identify variables that are called with ground terms or that returns a ground result - return pattern

3. Propagate bindings of a variable from left to right - calling pattern

Examples

```
set2list(phi, []).
set2list({X/T}, [X|L]) :- set2list(T, L).

perms(S) contains {L} :- set2list(S, L).



member(X, {X\_}).
diff(S1, S2) contains {X} :- member(X, S1), not member(
```

6. Experimental Results

6.1 Performance of SuRE

Performance of Memoization

```
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
```

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

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

distr(X, {L\_}) contains {[X|L]}.
```

	Size of argument set				
	3	4	5	6	7
with memo	0.0030	0.0102	0.0663	0.4389	13.59
without memo	0.0026	0.0115	0.1006	0.6157	15.36

Call-all Mode:

	Size of input set				
	3	4	5	6	7
with bags	0.0026	0.0115	0.1006	0.6157	15.36
with sets	0.0027	0.0149	0.1592	3.185	-

Call-one Mode:

	Size of argument set				
	3	4	5	6	7
with bag	0.0026	0.0127	0.1046	0.6083	10.92
with set	0.0028	0.0156	0.1496	2.844	127.8

Performance of Re-Do

# of nodes (# of edges)				
program	8(18)	16(38)	24(58)	32 (78)
`short1`	0.0032(2)	0.0303(6)	0.0979(10)	0.242(14)
`short2`	0.0072(3)	0.0156(6)	0.0238(9)	0.0324(12)

6.2 Comparisons with XSB and CORAL

XSB

- supports aggregations by two basic aggregate operators, bagReduce/4 and bagPO/3 and HiLog predicates.

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

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

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

	# of edges					
	18	38	58	78	138	495
SuRE/short2	0.0072	0.0156	0.0238	0.0324	0.0569	0.125
XSB/short2	0.02	0.021	0.02	0.02	0.02	0.059

CORAL

- needs different annotations depending on the aggregate operation to guarantee correct execution

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

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

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

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

end_module.
```

	# of edges			
	18	38	58	78
SuRE/short2	0.0072	0.0156	0.0238	0.0324
CORAL/short	0.02(0.06)	0.03(0.15)	0.07(0.29)	0.07(0.43)

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

7. Further Work

- Checking for duplicate elements when taking the union of the resulting sets
- Better representations for sets and better implementations for the primitive set operations.
- Efficient execution of functions whose result domain is totally ordered.
- Automatic Detection of lazy enumeration
- Garbage collection