BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-92-M7

"Contribution to Incremental Constraint Algorithms"

by

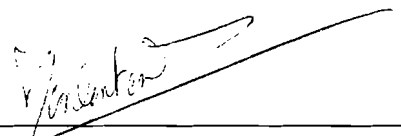
Ming-Li Cheng

Contribution to Incremental Constraint Algorithms

Ming-Li Cheng
Department of Computer Science
Box 1910 Brown University
Providence RI 02912
USA

April, 1992

This thesis by Ming-Li Cheng is accepted in its present form by the Department of Computer Science as satisfying the thesis requirement for the degree of Master of Science.



Prof. Pascal Van Hentenryck

Abstract

This paper introduces three problems which could occur in Constraint Logic Programming and presents possible ways to increase the efficiency of these problems. The first problem is enforcing arc consistency on a set of inequalities by reducing the domains of variables. The second problem is to find an incremental algorithm for least generalization in presence of on line bindings. The third problem is the extension of the second problem where each least generalization is required to be in a certain form. A dynamic algorithm is also generated for the problem.

Contents

Part I
Arc Consistency of Linear Inequalities

1 Introduction

The purpose of this article is to find an algorithm which enforces arc-consistency on a set of inequalities. The definition of arc-consistency for constraint is as the following:

Definition 1 Let $c(x_1, ..., x_n)$ be a constraint and D_i be the domain of x_i $(1 \leq i \leq n)$. Constraint $c(x_1, ..., x_n)$ is said to be *arc-consistent wrt* D_i $(1 \leq i \leq n)$ if the following condition holds for all i
• $\forall b_i \in D_i \; \exists b_1, ..., b_{i-1}, b_{i+1}, ..., b_n \in D_1, ..., D_{i-1}, D_{i+1}, ..., D_n$ such that $c(b_1, ..., b_n)$ holds.

This problem of inequalities is a typical Constraint Satisfaction Problem, which is defined by a finite set of variables taking values from finite domains and a set of constraints between these variables. The solution to this problem is an assignment of values to variables satisfying all inequalities. The technique is to reduce the search space by removing, from the domains of variables, values that cannot appear in the solution. The constraints in this paper are inequalities in the form of:

$$a_1x_1 + a_2x_2 + .. + a_nx_n + a \geq b_1y_1 + b_2y_2 + .. + b_my_m + b.$$

Where the a_i, b_j are constants, the x_i and y_j are variables with domains. Whether a constraint is satisfied depends on the maximum values of the variables on the left side and the minimum values of the variables on the right side.

To satisfy a constraint, the domains of variables on both sides of the constraint may be restricted by the constraint. Generally, the minimum value of each variable on the left side of an inequality may increase and the maximum value of each variable on the right side of an inequality may decrease. If there is only one inequality, each variable in the constraint should be considered once because the minimum value of a variable on the left side of the inequality depends on maximum values of all other variables on the left side and minimum values of all variables on the right side. Similarly, the maximum value of a variable on the right side of an inequality depends on minimum values of all other variables on the right side and all maximum values on the left side. Since the domains of the variables depend on each other, the variables in an inequality may need to be reevaluated if the domain of a variable in the inequality is changed by another constraint. We want to find a dynamic algorithm which reevaluates only those variables whose domains could possibly be changed and ignores all other variables. The following section gives a clear explanation for this problem.

2 Preliminaries and Basic Idea

In the following, for each variable v, we denote by $v.max$ the largest value in the domain of v and $v.min$ the smallest value in the domain of v.

for each constraint C_j:

$$a_1x_1 + a_2x_2 + .. + a_nx_n + a \geq b_1y_1 + b_2y_2 + .. + b_my_m + b.$$

We denote by $C_j.right$ the set of variables on the right side of C_j, $C_j.left$ the set of variables on the left side of C_j. $C_j.max$ is the maximum value obtained from the left side of the constraint; $C_j.max := a_1x_1.max + a_2x_2.max + .. + a_nx_n.max + a$, and $C_j.min$ is the minimum value obtained from the right side of the constraint; $C_j.min := b_1y_1.min + b_2y_2.min + .. + b_my_m.min + b$.

For any value u_i in the domain of a variable x_i on the left side, there must exist values in the domains of all other variables on the left side that satisfy :

$$a_iu_i + a_1u_1 + a_2u_2 + ... + a_{i-1}u_{i-1} + a_{i+1}u_{i+1} + ... + a_nu_n + a \geq C_j.min. \quad (1)$$

Otherwise u_i should be removed from the domain of x_i.

Similarly for any value u_i in the domain of a variable y_i on the right side, there must exist values in the domains of all other variables on the right side that satisfy :

$$\begin{aligned} & C_j.max \\ & \geq b_iu_i + b_1u_1 + b_2u_2 + ... + b_{i-1}u_{i-1} + b_{i+1}u_{i+1} + ... + b_mu_m + b. \end{aligned} \quad (2)$$

$$\begin{aligned} & a_iu_i + a_1x_1.max + ... + a_{i-1}x_{i-1}.max + a_{i+1}x_{i+1}.max + ... + a_nx_n.max + a \\ & \geq C_j.min. \end{aligned} \quad (3)$$

(3) is the special case of (1), which is sufficient to test the qualification of u_i. But, instead of testing each value in the domain of a variable, we derive a value, A_i, for x_i such that the minimum value of x_i cannot be smaller than A_i.

$$\begin{aligned} a_ix_i &\geq C_j.min - (a_1x_1.max + .. + a_{i-1}x_{i-1}.max \\ &\quad + a_{i+1}x_{i+1}.max + .. + a_nx_n.max + a). \\ \Longrightarrow a_ix_i &\geq C_j.min - (\sum_{k=1,k\neq i}^{n} a_k * x_k.max + a). \\ \Longrightarrow a_ix_i &\geq C_j.min - (C_j.max - a_i * x_i.max). \\ &\quad \text{we could derive } A_i \text{ from the inequality above.} \\ A_i &= \frac{C_j.min - (C_j.max - a_i * x_n.max)}{a_i}. \end{aligned}$$

Similarly, we could derive B_i for y_i such that the maximum value of y_i cannot be larger than B_i.

$$
\begin{aligned}
C_j.max &\geq b_i u_i + b_1 y_1.min + ..b_{i-1} y_{i-1}.min \\
&\quad + b_{i+1} y_{i+1}.min + ... + b_n y_n.min + b. \\
\Longrightarrow b_i y_i &\leq C_j.max - (\sum_{k=1, k\neq i}^{n} b_k * y_k.min + b). \\
\Longrightarrow b_i y_i &\leq C_j.max - (C_j.min - b_i * y_i.min). \\
&\quad \text{we could derive } B_i \text{ from the inequality above.} \\
B_i &= \frac{C_j.max - (C_j.min - b_i * y_i.min)}{b_i}.
\end{aligned}
$$

We could decide if the maximum value or minimum value of a variable needs to be changed by comparing it with B_i or A_i. If B_i is smaller than the maximum value of y_i on the right side, $\lfloor B_i \rfloor$ becomes the new maximum value of the variable. If A_i is larger than the minimum value of x_i on the left side, $\lceil A_i \rceil$ becomes the new minimum value of the variable. But, if B_i is smaller than $y_i.min$ or A_i is larger than $x_i.max$, there is no value in the domain of this variable could satisfy this inequality.

The following is the basic idea for updating the domains of variables in a constraint.

For each variable y on the left side
    Calculate B for y
    if $y.min < B < y.max$
        Update $y.max$
    else if $B < y.min$
        return FAILURE.

For each variable x on the right side
    Calculate A for x
    if $x.min < A < x.max$
        Update $x.min$
    else if $A > x.max$
        return FAILURE.

The above idea is enough when there is only one inequality. Now we are in position to discuss the system of several inequalities. An inequality needs to be reconsidered if the domain of one of the variables was affected by another inequality. Whenever $\lceil A_i \rceil$ becomes the new $x_i.min$, the constraints with the variable on the right side should be reconsidered since the minimums of these constraints will be higher.
Similarly, whenever $\lfloor B_i \rfloor$ becomes the new $y_i.max$, the constraints with the variable on the left side should be reconsidered since the maximums of these constraints will be lower.
The following is the basic idea of the algorithm for inequalities:

Step1 Initialize each inequality:
    Evaluate the maximum and minimum of the inequality.

$$C1 : v_1 + v_2 > v_3 + v_4$$
$$C2 : v_4 + v_5 > v_2 + v_6$$

Figure 1: Example 1

1.1 For each variable x on the left side of a constraint:
Evaluate A for x such that $x.min$ cannot be smaller than A.
Update the minimum value of x if necessary and keep all other constraints with the variable on their right side for further reevaluation.

1.2 For each variable y on the right side of a constraint:
Evaluate B for y such that $y.max$ cannot be larger than B.
Update the maximum value of y if necessary and keep all other constraints with the variable on their left side for further reevaluation.

Step2 For each constraint that needs to be reevaluated.
Update the max or min of the constraint.
Apply 1.1 or 1.2 for the variables in the constraint.

As we can see, the algorithm should be able to obtain the following information.

- Which constraints should be reconsidered?
- Which variable causes the constraint to be reconsidered?
- When a constraint C is reconsidered, how much should $C.max$ decrease or how much should $C.min$ increase?

3 Constraints for Reconsideration

Now let us take a look at Example 1. Assuming that the maximum value of v_2 is changed by $C2$, $C1$ should be reconsidered since v_2 is on its right side. Furthermore, if the maximum value of v_4 is changed when $C1$ is reconsidered, $C2$, which contains v_4 on its right side, should be considered again. The cycle will continue until the domains of v_2 and v_4 become stable.

A queue Q is used to keep the information of which constraints need to be reconsidered. Q is a set of tuples as (min, y, C, dv) such that $y \in C.right$ or as (max, x, C, dv) such that $x \in C.left$.

Whenever the maximum value of a variable, v, is decreased by some value dv, a tuple (max, v, C, dv) is inserted to Q for each constraint C with v on the left side of the constraint; similarly, whenever the minimum value of a variable, v, is increased

by some value dv, a tuple (min, v, C, dv) is inserted to Q for each constraint C with v on the right side of the constraint.

In Example 1, $(max, v_4, C2, d1)$ and $(min, v_2, C2, d2)$ are in Q if the domains of v_2 and v_4 change during the evaluation of $C1$. But, these tuples are redundant because $C2$ has not been evaluated yet, v_2 and v_4 have to be evaluated by $C2$ anyway.

We call the first evaluation of each constraint the initialization of the constraint. A tuple (m, v, C, dv) does not need to be inserted to Q if C is not yet initialized. $C.Init$ is set to TRUE after C is evaluated.

PushTuple(in v, m, dv, **inout** Q)
begin
case m **of**:
min:
$\mathrm{Q} := \mathrm{Q} \cup \{(min, v, C, dv) \mid \forall\ C$ such that $v \in C.right \wedge C.Init = \mathrm{TRUE}\}$;
max:
$\mathrm{Q} := \mathrm{Q} \cup \{(max, v, C, dv) \mid \forall\ C$ such that $v \in C.left \wedge C.Init = \mathrm{TRUE}\ \}$
end

DEQUEUE(Q):(m, v, C, dv)
Post: $(m, v, C, dv) \in \mathrm{Q}_0$ and $\mathrm{Q} = \mathrm{Q}_0 \setminus (m, v, C, dv)$.

In all specifications, we take the convention that a parameter p subscripted with 0 (i.e.,p_0) represents the value of p at call time.

4 Algorithm 1

We now present the algorithm for the problem. The algorithm initializes each constraint first; Each variable is evaluated in order that the maximum or minimum value of the variable is updated if necessary. $C.Init$ of the constraint is set to TRUE when the constraint is initialized. Second, for the constraint in each tuple of Q, reconsider the constraint and reevaluate its variables.

Calmaxmin(inout C):
$C.max := \sum_{l=1}^{n} a_i x_i.max + a$ where n is the number of variables on the left side of C.
$C.min := \sum_{l=1}^{m} b_i y_i.min + b$ where m is the number of variables on the right side of C.

CalB(in y, C, **out** B)
$B := \frac{C.max - (C.min - b * y.min)}{b}$ where b is the coefficient of y in C.

CalA(**in** x, C, **out** A)
$A := \frac{C.min-(C.max-a*x.max)}{a}$ where a is the coefficient of x in C;

Algorithm 1
begin
Q:= $\emptyset$;
for each constraint C **do**
ApplyInit(C, Q);
while Q $\neq \emptyset$ **do**
begin
tu := DEQUEUE(Q);
UpdateC(tu, Q)
end
end

ApplyInit(**inout** C, Q)
begin
Initialize(C);
for each $y \in C.right$ **do**
InitY(y, C, Q);
for each $x \in C.left$ **do**
InitX(x, C, Q)
end

Initialize(**inout** C)
begin
Calmaxmin(C);
$C.Init$:= TRUE
end

InitY(**inout** y, C, Q)
begin
CalB(y, C, B);
if $y.min < B < y.max$ **then**
begin
$dv := y.max - \lfloor B \rfloor$;
$y.max := \lfloor B \rfloor$;
PushTuple(y, max, dv, Q)
end
else if $B < y.min$

```
                return FAILURE
end


InitX(inout x, C, Q)
begin
        CalA(x, C, A);
        if x.min < A < x.max then
        begin
                dv := ⌈A⌉ − x.min;
                x.min := ⌈A⌉;
                PushTuple(x, min, dv, Q)
        end
        else if A > x.max
                return FAILURE
end


UpdateC(in (m, v, C, dv), inout Q)
begin
        case m of:
        max :
                C.max := C.max − a * dv where a is the coefficient of v in C;
        min :
                C.min := C.min + b * dv where b is the coefficient of v in C;
        for each x ∈ C.left do
                InitX(x, C, Q);
        for each y ∈ C.right do
                InitY(y, C, Q)
end
```

In the worst case, where each variable appears in all constraints, whenever the domain of a variable changes, all constraints have to be reconsidered and as we could see in procedure UpdateC, all variables are reevaluated by each constraint. Because a single domain update could trigger $n*e$ times of variable reevaluation and it is possible to have $n*d$ times of domain update, hence the time complexity of Algorithm 1 is $O(n^2ed)$; where n is the total number of variables, d is the largest domain size of all these variables and e is the number of inequalities.

5 Algorithm 2

In Algorithm 1, whenever a constraint is reconsidered, all its variables must be reevaluated by InitY or InitX. But it is not true that all variables have to change their

$C_1 : v + w + z >= q + 3$
$v : [0, 2]$
$w : [0, 1]$
$z : [0, 3]$
$q : [0, 4]$

Figure 2: Example 2

domains to satisfy the new status of the inequality.

> Consider Example 2, $C_1.min = 0 + 3 = 3$, $C_1.max = 2 + 1 + 3 = 6$, and the domains of v, w, z and q will not change. But, if $C_1.min$ changes from 3 to 5 because the minimum value of q changes from 0 to 2, the procedure UpdateC in Algorithm 1 will trigger the reevaluation of v, w and z by using $A_i = \frac{C_1.min - (C_1.max - a_i * x_i.max)}{a_i}$:
> $v : 0 < 5 - (6 - 1 * 2) = 1$, $v.min$ becomes 1.
> $w : 0 = 5 - (6 - 1 * 1) = 0$, $w.min$ remains 0.
> $z : 0 < 5 - (6 - 1 * 3) = 2$, $z.min$ becomes 2.
> Only v and z are affected by the change of $C_1.min$.

As we can see, when $C_1.min$ or $C_1.max$ changes, it will be nice if we could predict which variables need to be reevaluated.

Since $C.max$ can only decrease and $C.min$ can only increase, B_i will be smaller each time $C.max$ or $C.min$ changes; Similarly, A_i becomes larger each time $C.max$ or $C.min$ changes. The problem here is when will $\lfloor B_i \rfloor$ be small enough to replace $y_i.max$ or when will $\lceil A_i \rceil$ be large enough to replace $x_i.min$; $(B_i - x_i.max) * b_i$ and $(y_i.min - A_i) * a_i$ give the information we need. We denote by $old(C.max - C.min)$ the difference between $C.max$ and $C.min$ last time C was considered, and $new(C.max - C.min)$ the difference between $C.max$ and $C.min$ now.

> $y_i.max$ changes if $(old(C.max - C.min) - new(C.max - C.min))$
> $> (B_i - y_i.max) * b_i$.
> Also $x_i.min$ changes if $(old(C.max - C.min) - new(C.max - C.min))$
> $> (x_i.min - A_i) * a_i$.

If the new $\lfloor B_i \rfloor$ replaces the $x_i.max$ as the next $x_i.max$, the new B_i must be smaller by $(B_i - x_i.max)$; thus, if $x_i.max$ is fixed, $C.max - C.min$ should be $(B_i - x_i.max) * b_i$ smaller to make the new $\lfloor B_i \rfloor$ small enough to be the new $x_i.max$. If the amount of change of $C.max$ and $C.min$ each variable could tolerate in a constraint is provided, then, when $C.max$ or $C.min$ changes, only some variables in the constraint need to be reconsidered. We introduce another data structure, R, in order to support the information.

R = { $(C, v, dr) \mid v \in C.right \vee v \in C.left$ }. R is a set of tuples as (C, y, dr) or (C, x, dr) where $y \in C.right$, $x \in C.left$ and dr is the tolerant value of y or x such that $y.max$ and $x.min$ will not change unless the total amount of changes made to $C.max$ and $C.min$ is greater than dr.

$C.DIF$ is the total amount of changes made to $C.max$ and $C.min$. In Algorithm 2, whenever v_i in a constraint C is evaluated, a tuple $(v_i, C, (B_i - v_i.max) * b_i + C.DIF)$ is inserted to R if v_i is on the right side of C or $(v_i, C, (v_i.min - A_i) * a_i + C.DIF)$ is inserted if v_i is on the left side of C. When $C.DIF$ is $(B_i - v_i.max) * b_i$ or $(v_i.min - A_i) * a_i$ larger or more, the tuple will be retrieved and the reevaluation of v_i will be evoked.

5.1 Algorithm 2

```
Algorithm 2
begin
        Q := ∅; R := ∅;
        for each constraint C do
                ApplyInit(C, Q, R);
        while Q ≠ ∅ do
        begin
                tu := DEQUEUE(Q);
                UpdateC(tu, Q, R)
        end
end


ApplyInit(inout C, Q, R)
begin
        Initialize(C);
        for each y ∈ C.right do
                InitY(y, C, Q, R);
        for each x ∈ C.left do
                InitX(x, C, Q, R)
end


Initialize(inout C)
begin
        Calmaxmin(C);
        C.DIF := 0;
        C.Init := TRUE
end
```

```
InitY(inout y, C, Q, R)
begin
        CalB(y, C, B);
        if y.min < B < y.max then
        begin
                dv := y.max − ⌊B⌋;
                y.max := ⌊B⌋;
                PushTuple(y, max, dv, Q)
        end
        if B < y.min
                return FAILURE
        else
                R := R ∪ {(C, y, (B − y.max) * b + C.DIF)}
                        where b is the coefficient of y in C
end
```

```
InitX(inout x, C, Q, R)
begin
        CalA(x, C, A);
        if x.min < A < x.max then
        begin
                dv := ⌈A⌉ − x.min;
                x.min := ⌈A⌉;
                PushTuple(x, min, dv, Q)
        end
        if A > x.max
                return FAILURE
        else
                R := R ∪ {(C, x, (x.min − A) * a + C.DIF)}
                        where a is the coefficient of x in C
end
```

```
UpdateC(in (m, v, C, dv), inout Q, R)
begin
        case m of:
        max :
                C.max := C.max − a * dv where a is the coefficient of v in C;
                C.DIF := C.DIF + a * dv;
        min :
                C.min := C.min + b * dv where b is the coefficient of v in C;
                C.DIF := C.DIF + b * dv;
        InR := { (C, vr, dr) ∈ Q | (vr ∈ C.left ∨ vr ∈ C.right) ∧ dr < C.DIF};
```

```
        R := R \ InR;
        while InR ≠ ∅ do
        begin
                (C, vv, ddr) := DEqueueR(InR);
                if vv ∈ C.right then
                        InitY(vv, C, Q, R)
                else
                        InitX(vv, C, Q, R)
        end
end
```

DEqueueR(InR):(C, v, dr)
Post: $(C, v, dr) \in \text{InR}_0$ and $\text{InR} = \text{InR}_0 \setminus (C, v, dr)$.

As we can see in procedure UpdateC, InR collects the tuples of R with tolerant values smaller than $C.DIF$ since only the variables inside those tuples of C will be affected by the change of $C.max$ or $C.min$.

5.2 Examples

$c1 : x + y >= z + w$
$c2 : r + s >= t + y$
$x : [0, 5]$
$y : [0, 6]$
$z : [2, 4]$
$w : [2, 3]$
$r : [0, 5]$
$s : [2, 3]$
$t : [4, 5]$

Figure 3: Example 3

In Example 3, $c1.min = z.min + w.min = 4$ and $c1.max = x.max + y.max = 11$. $x.min$ remains 0 and $(c1, x, 2)$ is inserted to R because $A_x < x.min$ and $x.min - A_x = 2$. $y.min$ remains 0 and $(c1, y, 1)$ is inserted to R because $A_y < y.min$ and $y.min - A_y = 1$. $z.max$ remains 4 and $(c1, z, 5)$ is inserted to R because $B_z > z.max$ and $B_z - z.max = 5$. And $w.max$ remains 3 and $(c1, w, 5)$ is inserted to R because $B_w > w.max$ and $B_w - w.max = 6$.

$C : x + y + z >= w + 1$
$x : [0, 2]$
$y : [0, 1]$
$z : [0, 3]$
$w : [2, 4]$

Figure 4: Example 4

For $c2$, $c2.min = t.min + y.min = 4$ and $c2.max = r.max + s.max = 8$. $r.min$ becomes 1 and $(c2, r, 0)$ is inserted to R because $A_r > r.min$. $s.min$ remains 2 and $(c2, s, 3)$ is inserted to R because $A_s < s.min$ and $A_s - s.min = 3$. $y.max$ becomes 4 and $(c2, y, 0)$ is inserted to R because $y.max > B_y$. Since $c1$ contains y on its left side and $dv = 6 - 4 = 2$, $(max, y, c1, 2)$ is added to Q. And finally $t.max$ remains 5 and $(c2, t, 3)$ is inserted to R because $t.max < B_t$ and $B_t - t.max = 3$.

$c1$ is reconsidered when tuple $(max, y, c1, 2)$ is dequeued from Q; $c1.max = 11 - 2 = 9$ and $c1.min$ is still 4. Only one tuple in R, $(c1, y, 1)$, which contains $c1$ and which has tolerant value smaller than 2 needs to be reconsidered. Since $A_y = 4-(9-4) = -1$ and $y.min - A_y + C.DIF = 0-(-1)+2 = 3$, $y.min$ remains 0 and $(c1, y, 3)$ is inserted to R. Notice that $\lceil A_y \rceil$ was expected to replace $y.min$, but the increase of $y.max$ which occurred previously made $\lceil A_y \rceil$ too small to replace $y.min$. But, at least we could promise that the variables with larger tolerant values will not be reconsidered.

In example 4, $C.min = 2 + 1 = 3$ and $C.max = 2 + 1 + 3 = 6$. x remains 0 and $(x, C, 1)$ is inserted to R because $x.min > A_x$ and $x.min - A_x = 1$. y remains 0 and $(y, C, 2)$ is inserted to R because $y.min > A_y$ and $y.min - A_y = 2$. z remains 0 and $(z, C, 0)$ is inserted to R because $z.min = A_z$.

When $C.min$ becomes 5 because the minimum value of w changes from 2 to 4, we only need to reevaluate the tuples with tolerant value smaller than 2 (the change of $C.min$), which are $(z, C, 0)$ and $(x, C, 1)$. $x.max$ and $z.max$ will change and two proper tuples will be inserted to R.

Whenever $C.max$ or $C.min$ changes, each variable being evaluated is promised to be changed except those variables whose domains have been changed by other constraints since the last time the constraint was considered. As we could see in procedure UpdateC, InR collects variables with tolerant value smaller than the change of $C.max$ or $C.min$; thus, only limited number of variables will be reconsidered. In the worst case, where each variable appears in all constraints, whenever the domain of a variable changes, all constraints have to be reconsidered. Since there might be $n*d$ times of domain update, therefore the time complexity of Algorithm 2 is $O(ned)$; where n is the total number of variables, d is the largest domain size of all these variables and e is the number of inequalities.

$C : x + y + z >= w + q$
$x : [0, 2]$
$y : [0, 1]$
$z : [0, 3]$
$w : [1, 2]$
$q : [2, 3]$

Figure 5: Example 5

6 Algorithm 3

It is still possible to improve Algorithm 2 to make it work more efficient in some cases. Consider when $(max, y, c1, d1)$ and $(min, x, c1, d2)$ are in the queue. In Algorithm 2, if $(max, y, c1, d1)$ is dequeued first, $c1$ will be updated and may cause some new tuples to be inserted to Q. But after $(min, x, c1, d2)$ is dequeued, $c1$ needs to be updated again and may possibly insert more tuples to Q. It is better in this case to accumulate the changes of $C.max$ together with the changes of $C.min$. Instead of reconsidering some variables each time $C.max$ or $C.min$ is changed, we accumulate the changes of $C.max$ and $C.min$ until there is no tuple in Q which contains the constraint, and then reevaluate all the variables inside those tuples in R with tolerant values smaller than the accumulated value.

In example 5, $C.max = 2 + 1 + 3 = 6$, $C.min = 1 + 2 = 3$ and $(C, x, 1)$, $(C, y, 2)$, and $(C, z, 0)$ are in R. Suppose that $w.min$ increases to 2 and next $q.min$ increases to 3 due to other constraints, two tuples $(min, w, C, 1)$ and $(min, q, C, 1)$ are inserted to Q consecutively. If $(min, w, C, 1)$ is processed first, tuple $(C, z, 0)$ in R has to be reconsidered. The minimum value of z increases to 1, and tuple $(C, z, 1)((z.min - A_z) * a + C.DIF = 1)$ is inserted to R. When $(min, q, C, 1)$ is dequeued, not only $(C, x, 1)$ but also $(C, z, 1)$ has to be reconsidered; z has to be evaluated twice.

In this example, if we could delay the evaluation of variables until both $(min, w, C, 1)$ and $(min, q, C, 1)$ have been dequeued, some variables, like z, could be evaluated only once.

In procedure UpdateC of Algorithm 3, we test if there is any tuple in Q with constraint C and then decide whether to reevaluate the variables of C.

```
Algorithm 3
begin
        Q := ∅; R := ∅;
        for each constraint C do
                ApplyInit(C);
        while Q ≠ ∅ do
        begin
                tu := DEQUEUE(Q);
                UpdateC(tu, Q, R)
        end
end


ApplyInit(inout C, Q, R)
begin
        Initialize(C);
        for each y ∈ C.right do
                InitY(y, C, Q, R);
        for each x ∈ C.left do
                InitX(x, C, Q, R)
end


Initialize(inout C)
begin
        Calmaxmin(C);
        C.DIF := 0;
        C.Init := TRUE
end


InitY(inout y, C, Q, R)
begin
        CalB(y, C, B);
        if y.min < B < y.max then
        begin
                dv := y.max − ⌊B⌋;
                y.max := ⌊B⌋;
                PushTuple(y, max, dv, Q)
        end
        if B < y.min
                return FAILURE
        else
                R := R ∪ {(C, y, (B − y.max) * b + C.DIF)}
```

where b is the coefficient of y in C

end

InitX(inout x, C, Q, R)
begin
CalA(x, C, A);
if $x.min < A < x.max$ **then**
begin
$dv := \lceil A \rceil - x.min$;
$x.min := \lceil A \rceil$;
PushTuple(x, min, dv, Q)
end
if $A > x.max$
return FAILURE
else
R := R $\cup$ $\{(C, x, (x.min - A) * a + C.DIF)\}$
where a is the coefficient of x in C

end

UpdateC(in (m, v, C, dv), **inout** Q, R)
begin
case m **of**:
max :
$C.max := C.max - a * dv$ where a is the coefficient of v in C;
$C.DIF := C.DIF + a * dv$;
min :
$C.min := C.min + b * dv$ where b is the coefficient of v in C;
$C.DIF := C.DIF + b * dv$;
InQ := $\{ (m, vq, C, ddv) \in$ Q $\mid vq \in C.right \vee vq \in C.left \}$;
if InQ := $\emptyset$ **then**
begin
InR := $\{ (C, vr, dr) \in$ Q $\mid (vr \in C.left \vee vr \in C.right) \wedge dr < C.DIF\}$;
R := R $\setminus$ InR;
while InR $\neq \emptyset$ **do**
begin
(C, vv, ddr) := DEQUEUE(InR);
if $vv \in C.right$ **then**
InitY(vv, C, Q, R)
else
InitX(vv, C, Q, R)
end
end

end

The time complexity of Algorithm 3 is still *O(ned)*, only constant time better than Algorithm 2 in some cases. Since Algorithm 3 never pushs more tuples than Algorithm 2, it never works worse than Algorithm 2.

7 Conclusion

In this paper, we have introduced *O(ned)* dynamic algorithms for this problem of inequalities. But there is still room for some improvement. For example, when the sum of minimum values on the left side of an inequality is larger than the sum of maximum values on the right side of the inequality, this inequality will never need to be reconsidered again. That is to say, it is necessary to keep two more data for each inequality and the inequality could be satisfied forever as soon as the minimum value on the left side becomes larger than the maximum value on the right side. Though the time complexity is still *O(ned)*, it should be more efficient in some cases.

Part II
Incremental Term Generalization

1 Introduction

The idea of least generalization of terms or literals, which plays an important role in non-deductive reasoning, has been discussed in many papers including Plotkin [1970;1971], Reynolds [1970], Lassez, Maher and Marriott [1988], and Lassez and Marriott [1987]. These papers provide algorithms to find the least generalization of a set of terms or literals in time linear to the sum of sizes of these terms or literals if a proper data presentation is given.

Terms come in three varieties, constant symbols, variables and function applications. In this paper we restrict terms to the last two varieties. We say that the term $T1$ is more general than the term $T2$ if $T1\sigma = T2$ for some substitution σ. Substitution is a set of variable/term pairs and each variable/term pair is called a variable binding; each variable is said to be bound in this substitution. Applying a substitution to a formula means replacing each occurrence of a bound variable with its term. For example $P(v1, v2, v3)$ is more general than $P(v1, v2, f(v4))$ because there is a substitution $\sigma = \{v3/f(v4)\}$ such that $P(v1, v2, v3)\sigma = P(v1, v2, f(v4))$. A least generalization of a set of terms is a generalization which is less general than any other such generalization. For example, the least generalization of $\{P(x, y, f(x)), P(w, z, f(w))\}$ is $P(v1, v2, f(v1))$, but $P(v1, v2, f(v3))$ or $P(v1, v2, v3)$ is not the least generalization. There is only one least generalization for a set of terms.

Definition 1 A generalization of a set of terms $\{t1, t2, ..., tn\}$ is a term T such that there is a substitution σ_i for each t_i and $t_i = T\sigma_i$.

Definition 2 A least generalization of a set of terms $\{t1, t2, ..., tn\}$ is a generalization, T, such that for any other generalization of $\{t1, t2, ..., tn\}$, T', there is a substitution σ and $T = T'\sigma$.

But when a variable inside a term is bound, the generalizations relate to this term may need to be updated due to several different reasons. We have to know which generalizations need to be updated when a binding occurs, otherwise, we may need to regenerate all the generalizations all over again. The problem in this paper is not only to find the least generalization of any two terms, but also to attempt updating only those generalizations which need to be updated when a binding occurs. For example, the generalization of $(x, f(y))$ is $v1$ and the generalization of $(z, f(y))$ is $v2$. When x is bound to $f(x1)$, we want only the generalization of $(x, f(y))$ to be reconsidered but not the generalization of $(z, f(y))$. In fact, the generalization of $(f(x1), f(y))$ should be a functor f, like $f(v3)$.

Our aim in this paper is to find a dynamic algorithm which finds the least generalization of any two terms and is able to reconstruct only the generalizations which

need to be updated whenever a binding occurs.

2 Preliminaries

In this paper, we use $t1, t2, t3, tr, t1', t2', T1, T2, T3, ...$ for terms, $v, x, y, z, w, v1, v2, v3$, ... for variables, f, g, h, for functor names and P for predicate symbol.

Definition 3 A set of generalizations Φ is a set of tuples $< t1, t2, t3 >$ such that $t1, t2$ and $t3$ are terms and $t3$ is the least generalization of $t1$ and $t2$.

Definition 4 The domain of a set of generalizations Φ, noted $\text{dom}(\Phi)$, is the set of tuples $(t1, t2)$ for which there exists a $t3$ such that $< t1, t2, t3 > \in \Phi$.

Definition 5 A set of generalizations Φ is functional iff each $(t1, t2) \in \text{dom}(\Phi)$, there exists at most one $t3$ such that $< t1, t2, t3 > \in \Phi$.

In the following, we only consider functional sets of generalization and we note $\Phi(t1, t2)$, the unique $t3$, if it exists such that $< t1, t2, t3 > \in \Phi$.

In the following, we assume that we have an infinite set of variables that can be used during the generalization algorithm. A new variable can be obtained by calling the function NV, which returns a variable v not in use at the time of the call. Note that these variables are not accessible from the user and hence will never be bound externally.

The size of a term is the total number of terms in each level of the term. For example, the size of $f(x, f(y), h(f(y)))$ is 7 because the set of terms is $\{(x, f(y), y, h(f(y)), f(y), y, f(x, f(y), h(f(y))))\}$ in total.

The rest of the paper is organized in the following way. Section 3 describes the static algorithm for this problem. Section 4 presents a dynamic algorithm which maintains a set of generalizations may need to be updated when a binding occurs. Section 5 gives a dynamic algorithm for updating generalizations in the presence of a binding. Section 6 discusses the supporting data structures and their impact on time complexity.

3 The Static Algorithm for Generalization

3.1 Generalization of any Two Terms

There are four cases of least generalizations depending on the types of the two terms as depicted in Figure 1. Figure 2 depicts an example including all cases. In example 1, the generalization of $P(x, x, f(x), g(y, x))$ and $P(y, y, h(y), g(x, h(y)))$ is $P(v1, v1, v2, g(v3, v4))$. Each term in this generalization is obtained from the gener-

For the generalization of two terms $t1$ and $t2$, there are four cases:

- 1 If both $t1$ and $t2$ are variables, $\Phi(t1, t2)$ is a variable. For example, $\Phi(x, y) = v1$.
- 2 If one of $t1$ or $t2$ is a functor and the other is a variable, $\Phi(t1, t2)$ is a variable. For example, $\Phi(x, f(y)) = v2$.
- 3 If $t1$ and $t2$ are functors with different names, $\Phi(t1, t2)$ is a variable. For example, $\Phi(f(x), h(y)) = v3$.
- 4 If both $t1$ and $t2$ are functors with the same name and $t1 = f(t_1^1, ..., t_n^1)$ and $t2 = f(t_1^2, ..., t_n^2)$, $\Phi(t1, t2)$ is a functor with name f and arguments $\Phi(t_i^1, t_i^2)$, $1 \leq i \leq n$. $\Phi(t1, t2) = f(\Phi(t_1^1, t_1^2), ..., \Phi(t_n^1, t_n^2))$.

Figure 1: Four cases of least generalizations

$$P(x, x, f(x), g(y, x))$$
$$P(y, y, h(y), g(x, h(y)))$$

$v1 = \Phi(x, y)$
$v2 = \Phi(f(x), h(y))$
$v3 = \Phi(y, x)$
$v4 = \Phi(x, h(y))$
$g(v3, v4) = \Phi(g(y, x), g(x, h(y)))$
The generalization of the two predicates is:
$P(v1, v1, v2, g(v3, v4))$.

Figure 2: Example 1

alization of the corresponding terms in the two predicates; $v1$ is the generalization of x and y, $v2$ is the generalization of $f(x)$ and $h(y)$, $v3$ is the generalization of y and x, and $v4$ is the generalization of x and $h(y)$. $v1$ is given as the least general term of $\{x, y\}$ because x and y are two variables. For the second column, which is also $\{x, y\}$, we should be able to recognize that $v1$ has been given as their least general term and avoid generating a new term. The third column, $\{f(x), h(y)\}$, contains functors with different function names. Since neither h, f; nor any other function name can represent these terms, the least general term should be a variable, say $v2$. The fourth column, $\{g(y, x), g(x, h(y))\}$, contains terms with the same function name g, a variable is not enough to be their least general term. The least generalization should also begin with g and its two subterms should be the least general terms of $\{y, x\}$ and $\{x, h(y)\}$. Notice that although $\{y, x\}$ contains exactly the same terms as $\{x, y\}$, their least general terms are different because they are in different order. $v3$ is given as the general term of $\{y, x\}$, $v4$ is given as the general term of $\{x, h(y)\}$ and finally $g(v3, v4)$ is created as the generalization of $\{g(y, x), g(x, h(y))\}$.

3.2 The Static Algorithm

The static generalization algorithm can now be defined as follows.
Generalize($t1, t2$):
Given two terms $t1, t2$, Generalize($t1, t2$) returns a set of generalizations, Φ, such that $\Phi(t1, t2)$ is the least generalization of $(t1, t2)$. We denote by functor(t) the function name of the functor t, type(t) the type of term t.

Generalize($t1, t2$)
begin
    $\Phi := \emptyset$;
    Generalize-aux($t1, t2, \Phi$)
end

Generalize-aux(in $t1, t2$, **inout** Φ)
begin
    if $(t1, t2) \notin$ dom(Φ) **and** $t1 \neq t2$ **then**
    begin
        if type($t1$) = type($t2$) = variable **then**
            $\Phi := \Phi \cup \{< t1, t2, NV >\}$
        else if type ($t1$) = variable **then**
            $\Phi := \Phi \cup \{< t1, t2, NV >\}$
        else if type($t2$) = variable **then**
            $\Phi := \Phi \cup \{< t1, t2, NV >\}$
        else if functor($t1$) $\neq$ functor($t2$) **then**
            $\Phi := \Phi \cup \{< t1, t2, NV >\}$
        else

```
            begin
                  let
                  t1 = f(t1_1, ..., t1_n);
                  t2 = f(t2_1, ..., t2_n);
                  in
                  for i := 1 to n do
                        Generalization-aux(t1_i, t2_i, Φ);
                  Φ := Φ ∪ {< t1, t2, f(Φ(t1_1, t2_1), ..., Φ(t1_n, t2_n)) >}
            end
      end
end
```

The algorithm above finds the generalization of any two terms. It can be generated easily to apply to an arbitrary set of terms. The basic idea is that it is possible to obtain the generalization of a set of terms $\{t1, ..., tm\}$ by performing the least generalizations of pairs, e.g $glb(..., (glb(glb(t1, t2), glb(t3, t4)), ..))$. A divide and conquer approach is used to organize the generalizations in the form of a binary tree. As we could see in Figure 3, several generalizations are produced as the footsteps to the generalization of the whole. This approach can be used as the basic step of an algorithm receiving any number of terms. Suppose that the basic step receives terms $\{x, h(y), f(w), f(z)\}$. The basic step first uses the algorithm above to produce the generalizations of $(x, h(y))$ and $(f(w), f(z))$. And it produces $v1 = \Phi(x, h(y))$ as the generalization of the first pair and $f(v2) = \Phi(f(w), f(z))$ as the generalization of the second pair. Then, the basic step continues calling the algorithm for $v1$ and $f(v2)$, $v3$ is produced as the generalization of $v1$ and $f(v2)$. Since there is no other generalization in this level, $v3$ becomes the generalization of the four terms.

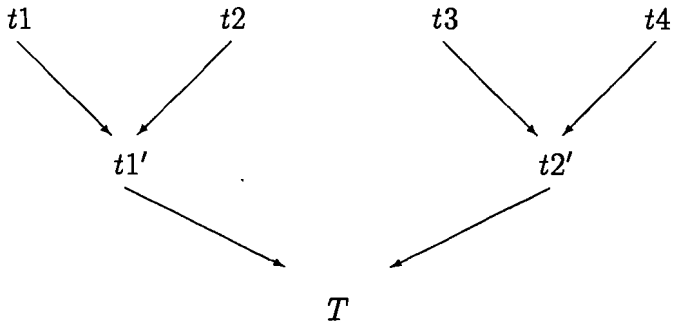


Figure 3: A divide and conquer approach is used for the generalization of $\{t1, t2, t3, t4\}$

4 A New Generalization Algorithm

Our main interest is to design an algorithm that would update the generalizations dynamically in the presence of new bindings.

In the static algorithm, when a binding occurs, we should either do the generalizations all over again, or scan through all the general terms to find out which ones need to be regenerated. A general term that is generated from a term which is or which contains the term being bound has to be reconsidered. But, for dynamic updating, it would be more appropriate to keep more information during the process of generalization. We should be able to know which generalizations may be reconsidered when bindings occur.

We now generalize the dynamic algorithm to produce, in addition to the set Φ, a list of the least generalizations $(t1, t2, tr)$ where tr is the least generalization of $t1, t2$ which may need to be reconsidered when a binding occurs. The generalization is straightforward and simply returns an element for each of the basic cases.

Generalize(inout $t1, t2$)
begin
    Q := $\emptyset$;
    $\Phi := \emptyset$;
    Generalize-aux($t1, t2, \Phi$, Q)
end

Generalize-aux(in $t1, t2$, **inout** Φ, Q)
begin
    if $(t1, t2) \notin$ dom(Φ) **and** $t1 \neq t2$ **then**
    begin
        if type($t1$) = type($t2$) = variable **then**
            Update(Φ, Q, $t1, t2$)
        else if type($t1$) = variable **then**
            Update(Φ, Q, $t1, t2$)
        else if type($t2$) = variable **then**
            Update(Φ, Q, $t1, t2$)
        else if functor($t1$) $\neq$ functor($t2$) **then**
            Update(Φ, Q, $t1, t2$)
        else
        begin
            let
            $t1 = f(t_1^1, ..., t_n^1)$;
            $t2 = f(t_1^2, ..., t_n^2)$;
            in
            for $i := 1$ to n **do**
                Generalization-aux(t_i^1, t_i^2, Φ, Q);
            $\Phi := \Phi \cup \{< t1, t2, f(\Phi(t_1^1, t_1^2), ..., \Phi(t_n^1, t_n^2)) >\}$

```
            end
        end
end

Update(inout Φ , Q, in t1, t2)
begin
                    v := NV;
                    Φ := Φ ∪ {< t1, t2, v >};
                    Q := Q ∪ {(t1, t2, v)}
end
```

The kind of generalization that needs to be collected for reconsideration is the one with variable as generalization. For generalization like $f(z) = \Phi(f(x), f(y))$, there must also exist $z = \Phi(x, y)$. A binding to x or y may affect z, but there is no direct affect on $f(z)$. All $f(x), f(y)$ and $f(z)$ are terms bound to functor f, no binding could occur to them directly. Hence only generalizations with variable as generalization need to be reconsidered and they are collected to Q as we could see in the algorithm.

Q is a set of tuples $(t1, t2, tr)$ that may need to be reconsidered when bindings occur. Whenever a generalization is generated except in the case that the generalization is a functor, an element is added to Q.

The set Q produced by Generalization for $\{x, y, g(x), f(z), x, g(z), g(x), g(z)\}$ is depicted in Figure 4 (They are organized by the basic step that we have discussed in the previous section).

5 Update Generalization

Whenever a binding occurs, there may be some generalizations that need to be updated; When x is bound to t, all the tuples $(t1, t2, tr)$ in Q, such that where $x \in t1$ or $x \in t2$ should be reconsidered. As we could see in Figure 4, $(x, y, v1)$, $(g(x), f(z), v2)$, $(x, g(z), v3)$ and $(x, z, v4)$ will be affected if x is bound to another term, but only $(x, y, v1)$ will be affected if y is bound.

The tuples to be reconsidered could be divided in several cases as depicted below. Let $(t1, t2, tr)$ denotes the tuple with x inside and $O[x/t]$ denotes the object O where each occurrence of x has been replaced by t. In the following, $t1' = t1[x/t]$ and $t2' = t2[x/t]$.

- If $(t1', t2') \in \text{dom}(\Phi)$, the generalization of $t1$ and $t2$, tr, should be bound to $\Phi(t1', t2')$.
- If $(t1', t2') \notin \text{dom}(\Phi)$ and both $t1'$ and $t2'$ become functors with the same name, a new generalization should be generated for $t1'$ and $t2'$ and the old generalization, tr, of $t1$ and $t2$, should be bound to the new generated term.

- If $(t1', t2') \notin \mathrm{dom}(\Phi)$ and $t1'$ and $t2'$ are functors with different names or any one of $t1'$ or $t2'$ is a variable, since the generalization remains a variable, tr could remain unchanged; $(t1, t2, tr)$ in Q is replaced by $(t1', t2', tr)$.

As we can see, the behavior of regeneralization includes checking the domain of Φ, generating more general terms, and binding one general term to another general term.

5.1 Algorithm for Regeneralization

We are now in position to present the algorithm to update the generalizations in the case of a binding. First, all the tuples with the term being bound inside are collected from Q. Second, each of these tuples is examed and applied with different approach as depicted above. We note VAR(t) the set of variables in term t.

UpdateGeneralization(inout Φ, Q, **in** x, t)
begin
    R := { $(t1, t2, tr) \in$ Q | $x \in t1 \vee x \in t2$};
    Q := Q \ R;
    R := R[x/t];
    PropagateBinding(R, Q, Φ)
end

PropagateBinding(inout R, Q, Φ)
begin
    while R $\neq \emptyset$ **do**
    begin
        tu := DEQUEUE(R);
        Regenerate(tu, Q, Φ)
    end
end

Regenerate(in $(t1', t2', tr)$, **inout** Q, Φ)
begin
    if $(t1', t2') \in \mathrm{dom}(\Phi)$ **then**
        UpdateGeneralization(Φ, Q, tr, $\Phi(t1', t2')$)
    else if type($t1'$) = functor **and** type($t2'$) = functor
        and functor($t1'$) = functor($t2'$) **then**
    begin
        Generalize-aux($t1', t2', \Phi$, Q);
        UpdateGeneralization(Φ, Q, tr, $\Phi(t1', t2')$)

end
else
$$Q := Q \cup \{(t1', t2', tr)\}$$
end

DEQUEUE(R):$(t1, t2, tr)$
Post: $(t1, t2, tr) \in R_0$ and $R = R_0 \setminus (t1, t2, tr)$.

5.2 An Example

Consider Figure 4 again. Tuples $(x, y, v1)$, $(g(x), f(z), v2)$, $(x, g(z), v3)$, and $(x, z, v4)$ are in R if x is bound to $g(w)$.

For $(x, y, v1)$, which becomes $(g(w), y, v1)$, there is no general term, which should be a variable, for $(g(w), y)$. Hence $v1$ remains unchanged. Similar to $(x, y, v1)$, $(g(x), f(z), v2)$ becomes $(g(g(w)), f(z), v2)$ and $v2$ remains unchanged. For $(x, g(y), v3)$, which becomes $(g(w), g(y), v3)$, a new least general term should be generated because $v3$ could no longer qualify to be the least general term of $g(w)$ and $g(y)$. Suppose that $g(v8)$ is generated as the least general term of $g(w)$ and $g(y)$, $v3$ should be bound to $g(v8)$ by calling UpdateGeneralization recursively. The binding of $v3$ to $g(v8)$ makes $(v3, g(v4), v6)$ become $(g(v8), g(v4), v6)$ and a new generated term $g(v9)$ is given as the generalization of $g(v8)$ and $g(v4)$; therefore, $v6$ should be bound to $g(v9)$. The only generalization relates to $v6$, $(v5, v6, v7)$, becomes $(v5, g(v9), v7)$. For $(x, z, v4)$, which becomes $(g(w), z, v4)$, $v4$ should remains unchanged. Figure 5 is the status of Q after the binding of x.

6 Supporting Data Structures

Some operations in the algorithm are crucial to the timing of the problem. To obtain an efficient algorithm, some proper data structures must be chosen. The following are the operations in our consideration.

1 Binding Terms.

2 The operations on Φ, the set of generalizations.

1. Checking the domain of Φ.
2. Adding a new element to Φ.
3. Retrieving a element of Φ.

3 Retrieving the tuples in Q which need to be reconsidered when a variable is bound.

1 $(x, y, v1)$
2 $(g(x), f(z), v2)$
3 $(x, g(z), v3)$
4 $(x, z, v4)$
5 $(v1, v2, v5)$
6 $(v3, g(v4), v6)$
7 $(v5, v6, v7)$.

Figure 4: Q produced by Generalization for $\{x, y, g(x), f(z), x, g(z), g(x), g(z)\}$

1 $(g(w), y, v1)$
2 $(g(g(w)), f(z), v2)$
3 $(w, z, v8)$
4 $(g(w), z, v4)$
5 $(v1, v2, v5)$
6 $(v8, v4, v9)$
7 $(v5, g(9), v7)$.

Figure 5: The Q set after $x = g(w)$

6.1 Binding Terms

When a term is bound to another, all the terms which contain this term will be affected. For example, when x is bound to $f(x1)$, $f(x)$ and $g(y, f(x))$ will become $f(f(x1))$ and $g(y, f(f(x1)))$. To avoid updating all the related terms whenever a binding occurs, we should choose a right representation for terms.

A term could be represented by a name, which is either a variable name or a function name, and a set of pointers to the subterms of the term if the term is a functor. For instance, $g(x, y)$ is represented by name g and pointers to x and y. In addition to the name and pointers to subterms, a term could also have a pointer to the term it is bound to; when x is bound to $f(x1)$, the pointer of x will point to $f(x1)$. All the terms relate to x, like $f(x)$ and $g(y, f(x))$, will be automatically taken care of because they have pointers point to x (directly or through other terms) and x has pointed itself to $f(x1)$. No individual effort is needed for each of these terms.

6.2 The Timing of Generalization

We could use a hash table to represent the generalization set, Φ. Suppose that HF is the hash function for this table, HF$(t1, t2)$ leads to a location in the hash table where a pointer set containing pointers to $t1$, $t2$ and $t3$ will be put to represent the tuple $< t1, t2, t3 >$ in Φ.

A single location in the hash table may contain several pointer sets, each pointer set $(pt1, pt2, pt3)$ such that $pt1$ is the pointer to $t1$, $pt2$ is the pointer to $t2$ and $pt3$ is the pointer to $t3$ and $t3$ is the generalization of $t1$ and $t2$.

To know if there is a generalization for $t1$ and $t2$ (if $(t1, t2) \in \text{dom}(\Phi)$), we simply check if there is a pointer set containing pointers to $t1, t2$ and the generalization of them in the location of the hash table assigned to by HF$(t1, t2)$.

Similarly, to retrieve the generalization of $t1$ and $t2$, the location in the hash table assigned to by HF$(t1, t2)$ is searched until a pointer set containing pointers to $t1, t2$ and their generalization is found. We follow the third pointer and reach the generalization of $t1$ and $t2$.

The above two operations on Φ require time proportional to the size of a term because we have to follow pointer to the term and scan through it to see if it is the $t1$ or $t2$ we are looking for.

To add a tuple $< t1, t2, t3 >$ to Φ, we simply add a pointer set which contains pointers to $t1, t2$ and $t3$ to the location in the hash table assigned to by HF$(t1, t2)$.

6.3 Direct Accessing for Term Updates

In UpdateGeneralization, when a binding like $x = t$ occurs, we need to retrieve those tuples in Q with x inside: R = $\{ (t1, t2, tr) \in \text{Q} \mid x \in t1 \vee x \in t2 \}$.

Without any assistance, we have to scan through each tuple to see if it contains x, which requires time linear to the size of Q. By this way, no matter how many generalizations are actually affected, the time complexity will always be *O(sN)* where

N is the total number of generalizations and s is the size of a term. But, if we could provide knowledge of which tuples will be affected by the binding of a variable, these tuples could be accessed directly without checking all the generalization tuples.

To support direct accessing, each variable in the symbol table maintains a set of pointers and each pointer points to a tuple in Q that could be affected by the variable. When a binding occurs, all the generalizations produced from terms containing the variable being bound are affected, hence, whenever a generalization tuple is added to Q, the pointer of the tuple should be given to all the variables inside the two terms of the generalization. For example, when $v1 = \Phi(h(x,y),w)$ is generated and $(h(x,y),w,v1)$ is added to Q, the pointer to $(h(x,y),w,v1)$ should be added to the pointer sets of x, y and w. It is to say we have to scan through the two terms to obtain all the variables inside. Thus, the timing of Generalize-aux is proportional to the larger size of these two terms.

As we could see in Regenerate, time to regenerate a single general term is still proportional to the size of a term because we first have to make sure the generalization of the updated terms does not exist and then call Generalize-aux to obtain the new generalization. Both these two steps take time linear to the size of a term. But, the reconsideration of a generalization may bind a general term to another general term and then trigger more generalizations and bindings.

7 Conclusion

In our algorithm the time to generate the least general term of two terms is proportional to the larger size of the two terms since the information for term updating must be kept during generalization.

Updating of one general term requires time linear to the size of a term because scanning through a term may be required by comparing terms or generating new general terms.

But, exactly how much time is needed to update generalizations because of a single term binding? In the worst case, all the general terms will be affected by a single binding. Suppose that the total number of general terms is N and the maximum size of a term is limited to s. This is still a *O(sN)* algorithm although in many cases it is *O(sM)* such that M is the total number of generalizations updated and $M < N$.

Part III
Restricted Least Generalization

We have introduced a least generalization algorithm before, now we would like to introduce a problem where the least generalization is required to be in a certain form. For example, the generalization of $f(x)$ and $f(y)$ in the form of $f(h(z))$ is $f(h(v1))$. This problem may occur when the generalization of any two terms is bound to a certain form.

1 Introduction

The problem can be defined as: Given $t1$, $t2$ and a form t, we want to find T, the least generalization of $t1$ and $t2$ with respect to t. The following is the definition of the generalization. We denote by $glb(t1, t2)$ the least generalization of $t1$ and $t2$, $mgu(t1, t)$ the most general unification of $t1$ and t.

Definition 1 The least generalization of $t1$ and $t2$ with respect to t, noted $rglb(t1, t2, t)$, is T

$$\begin{aligned} iff \quad & \exists\Theta\ t1\Theta = t\Theta \wedge \exists\Theta'\ t2\Theta' = t\Theta' \wedge t' = glb(t1, t2) \wedge T = mgu(t', t) \ \vee \\ & \exists\Theta\ t1\Theta = t\Theta \wedge \neg\exists\Theta'\ t2\Theta' = t\Theta' \wedge T = mgu(t1, t) \ \vee \\ & \neg\exists\Theta\ t1\Theta = t\Theta \wedge \exists\Theta'\ t2\Theta' = t\Theta' \wedge T = mgu(t2, t). \end{aligned}$$

It is not defined otherwise.

Before we discuss the problem, it is necessary to know that a term $t1$ is compatible with a term t if there is a substitution Θ such that $t1\Theta = t\Theta$.

Remember the least generalization problem we have before, for any two terms we could find a proper term as their least generalization. But, suppose now the general term is bound to a certain term or it is required to be in a certain form, the original least generalization may not be able to satisfy this situation. For example, the least generalization of $f(x)$ and $h(y)$ is v. If v is bound to $f(v1)$, there is no generalization of $f(x)$ and $h(y)$ could be in the form of $f(v1)$ because $h(y)$ is not compatible with $f(v1)$; thus, $h(y)$ should be excluded and moreover $f(x)$ and $f(v1)$ should be unified and be the generalization of itself. The following are the different cases of generalization of $t1$ and $t2$ with respect to t:

- If both $t1$ and $t2$ are compatible with t, the generalization of $t1$ and $t2$ should be the unification of the least generalization of $(t1, t2)$ and t.

  For example, the generalization of $f(x)$ and $f(y)$ with respect to $f(h(z))$ is Unify(Generalization($f(x), f(y)$), $f(h(z))$) = Unify($f(v), f(h(z))$) = $f(h(z))$.

- If $t1$ is compatible with t but $t2$ is not, the generalization is the unification of $t1$ and t.

  For example, the generalization of $f(x)$ and $h(y)$ with respect to $f(f(z))$ is Unify($f(x), f(f(z))$) = $f(f(z))$.

- If $t2$ is compatible with t but $t1$ is not, the generalization is the unification of $t2$ and t.
- If both $t1$ and $t2$ are not compatible with t, no generalization could be obtained.

Now we define a data structure, Δ, which will be used in our algorithm.

Definition 2 A set of generalizations Δ is a set of tuples $< t1, t2, t, T >$ such that T is the least general term of $t1$ and $t2$ with respect to t.

Definition 3 The domain of a set of generalizations Δ, noted $\mathrm{dom}(\Delta)$, is the set of tuples $(t1, t2, t)$ for which there exists a T such that $< t1, t2, t, T > \in \Delta$.

Definition 4 A set of generalizations Δ is functional iff each $(t1, t2, t) \in \mathrm{dom}(\Delta)$, there exists at most one T such that $< t1, t2, t, T > \in \Delta$.

In the following, we only consider functional sets of generalization and we note $\Delta(t1, t2, t)$, the unique T, if it exists such that $< t1, t2, t, T > \in \Delta$.

The rest of the paper is organized as follows. Section 2 presents the static algorithm for this problem. Section 3 presents a dynamic algorithm for this problem which generates a set of generalizations that may need to be reconsidered.

2 The Static Algorithm

The static generalization algorithm can now be defined as follows.
GeneralizeF$(t1, t2, t)$:
Given two terms $t1, t2$ and a form t, GeneralizeF$(t1, t2, t)$ returns a set of generalizations Δ such that $\Delta(t1, t2, t)$ is the generalization of $(t1, t2)$ with respect to t.

First let us assume that Unify$(T, T1)$ makes T and $T1$ the same term and returns the term itself.

This algorithm is based on the different cases discussed previously, but here we find out whether the form t is a variable before checking the compatibility of the two terms with the form. If the form t is a variable, no compatibility needs to be checked because a variable is compatible with any variable or functor.

GeneralizeF(inout $t1, t2, t$)
begin
    $\Delta := \emptyset$;
    GeneralizeF-aux$(t1, t2, t, \Delta)$
end

```
GeneralizeF-aux(in t1, t2, t, inout Δ)
begin
        if (t1, t2, t) ∉ dom(Δ) then
        begin
                if type(t) = variable then
                begin
                        Generalize(t1, t2);
                        Δ := Δ ∪ {< t1, t2, t, Φ(t1, t2) >}
                end
                else if Compatible(t1, t) ∧ Compatible(t2, t) then
                begin
                        Generalize(t1, t2);
                        T := Unify(Φ(t1, t2), t);
                        Δ := Δ ∪ {< t1, t2, t, T >}
                end
                else if Compatible(t1, t) ∧ ¬ Compatible(t2, t) then
                begin
                        T := Unify(t1, t);
                        Δ := Δ ∪ {< t1, t2, t, T >}
                end
                else if ¬ Compatible(t1, t) ∧ Compatible(t2, t) then
                begin
                        T := Unify(t2, t);
                        Δ := Δ ∪ {< t1, t2, t, T >}
                end
                else if ¬ Compatible(t1, t) ∧ ¬ Compatible(t2, t) then
                        return FAILURE
        end
end


Compatible(in t1, t2):Bu
begin
        Θ := ∅;
        return Comp-aux(t1, t2, Θ)
end

Comp-aux(in t1, t2, inout Θ):Bu
begin
        if type(t1) = variable
                if t1 ∈ VAR(t2)
                        return FAILURE
                else
                        Θ := {(t1/t2)}
```

```
        else if type(t2) = variable
                if t2 ∈ VAR(t1)
                        return FAILURE
                else
                        Θ := {(t2/t1)}
        else if functor(t1) ≠ functor(t2) then
                return FAILURE
        else
        begin
                let
                t1 = f(t1_1, ..., t1_n);
                t2 = f(t2_1, ..., t2_n);
                in
                for i := 1 to n
                begin
                        if (Comp-aux(t1_i, t2_i, Θ)) = FAILURE then
                                return FAILURE
                        else
                        begin
                                t1 := t1Θ;
                                t2 := t2Θ
                        end
                end
        end
        return TRUE;
end
```

3 The Dynamic Algorithm

As we could see in the static algorithm, in the case where only $t1$ is compatible with t, the generalization is the unification of $t1$ and t. Since $t1$ and t become the same term as well as the generalization after the unification, whenever $t1$ or t changes, the generalization changes automatically. It is similar in the case where only $t2$ is compatible with t. The generalizations produced in these two cases will never need to be reconsidered because they will be taken care of automatically. But, in the case where both $t1$ and $t2$ are compatible with t, the generalization of $t1$ and $t2$ remains open to be affected; Thus, these generalizations should be kept for reconsideration in the future. We will use a queue QU containing elements of the form $(t1, t2, t, T)$, informally $(t1, t2, t, T) \in$ QU if T is the generalization of $t1$ and $t2$ with respect to t and both $t1$ and $t2$ are compatible with t. The dynamic generalization algorithm can

now be defined as follows.
GeneralizeF($t1, t2, t$):
Given two terms $t1, t2$ and a form t, GeneralizeF($t1, t2, t$) returns a set of generalizations Δ such that $\Delta(t1, t2, t)$ is the generalization of $(t1, t2)$ with respect to t, and QU, a set of generalizations $(t1, t2, t, T)$ such that T is the least generalization of $t1, t2$ with respect to t which may need to be reconsidered when bindings occur.

GeneralizeF(inout $t1, t2, t$)
begin
    QU := $\emptyset$;
    $\Delta := \emptyset$;
    GeneralizeF-aux($t1, t2, t, \Delta$, QU)
end

GeneralizeF-aux(in $t1, t2, t$, **inout** Δ, QU)
begin
    if ($t1$, $t2, t$) $\notin$ dom(Δ) **then**
    begin
        if type(t) = variable **then**
        begin
            Generalize($t1, t2$);
            $\Delta := \Delta \cup \{< t1, t2, t, \Phi(t1, t2) >\}$
        end
        else if Compatible($t1, t$) $\wedge$ Compatible($t2, t$) **then**
        begin
            Generalize($t1, t2$);
            T := Unify($\Phi(t1, t2), t$);
            Update(Δ, QU, $t1, t2, t, T$)
        end
        else if Compatible($t1, t$) $\wedge$ $\neg$ Compatible($t2, t$) **then**
        begin
            T := Unify($t1, t$);
            $\Delta := \Delta \cup \{< t1, t2, t, T >\}$
        end
        else if $\neg$ Compatible($t1, t$) $\wedge$ Compatible($t2, t$) **then**
        begin
            T := Unify($t2, t$);
            $\Delta := \Delta \cup \{< t1, t2, t, T >\}$
        end
        else if $\neg$ Compatible($t1, t$) $\wedge$ $\neg$ Compatible($t2, t$) **then**
            return FAILURE
    end
end

Update(inout Δ, QU, **in** $t1, t2, t, T$)
begin
$\Delta := \Delta \cup \{< t1, t2, t, T >\}$;
QU := QU $\cup \{(t1, t2, t, T)\}$
end

4 Update Generalization

Now we want to generate an algorithm to update the generalizations in the presence of a binding; When a term is bound to another term, all the generalizations relate to this term should be reconsidered. For example, the generalization of $f(x)$ and y with respect to z is v, but if y is bound to $h(y1)$, $h(y1)$ is not compatible with z; thus, the generalization should be Unify$(f(x), z) = f(x)$.

When a binding, $x = x1$, occurs, a tuple $(t1, t2, t, T)$ in QU has to be reconsidered either because $x \in t1$ or $x \in t2$ or $x \in t$. The updated $t1, t2$ and t after the binding actually become $t1', t2'$ and t' such that $t1' = t1[x/x1]$, $t2' = t2[x/x1]$ and $t' = t[x/x1]$; $O[x/x1]$ denotes the object O where each occurrence of x has been replaced by $x1$.

We have to reconsider the generalization of $(t1', t2', t)$ by calling GeneralizeF-aux again.

Now we present the algorithm for generalization update. First, all the tuples containing the term being bound are collected from QU. Second, each tuple is reconsidered by GeneralizeF-aux. And also the produce of a new general term due to a binding may lead to another term binding; The old general term should be bound to the new one.

UpdateGeneralizationF(inout Δ, QU, **in** $x, x1$)
begin
R := { $(t1, t2, t, T) \in$ QU | $x \in t1 \vee x \in t2 \vee x \in t$};
QU := QU \ R;
R := R$[x/x1]$;
PropagateBinding(R, QU, Δ)
end

PropagateBinding(inout R, QU, Δ)
begin
while R $\neq \emptyset$ **do**
begin
$(t1', t2', t', T)$:= DEQUEUE(R);
GeneralizeF-aux$(t1', t2', t', \Delta$, QU);
if $T \neq \Delta(t1', t2', t')$ **then**
UpdateGeneralizationF(Δ, QU, T, $\Delta(t1', t2', t')$)

end
end

DEQUEUE(R):$(t1, t2, t, T)$
Post: $(t1, t2, t, T) \in \mathrm{R}_0$ and $\mathrm{R} = \mathrm{R}_0 \setminus (t1, t2, t, T)$.

5 Time Complexity

In GeneralizeF-aux, the key operations are Compatible, Generalize and Unify. Generalize is the same one we introduced before, which takes time proportional to the size of a term to produce the least generalization of any two terms. It is obvious as we read the Compatible algorithm that it also takes time proportional to the size of a term to decide if two terms are compatible with each other. And also Unify may require the same time complexity because we may need to scan through the two terms to be unified. All the three key operations take time proportional to the size of a term; thus, we could conclude that the time to generate a generalization of any two terms with respect to a form is proportional to the size of a term.

Because GeneralizeF-aux is also called when a generalization needs to be changed, the time to update a single generalization is also linear to the size of a term. But, the update of a generalization might lead to more bindings because the old generalization needs to be bound to the new generalization; thus, it is possible in the worst case that a single binding affects all the generalizations. The total time of processing a single binding is $O(sN)$ where N is the total number of general terms and s is the maximum size of a term.

A An Application on Least Generalization

Here we introduce an application on least generalization. Lex is a program that receives a number of predicates and generates the least generalization for them. In the program, a divide and conquer approach is used to organize the generalizations in the form of a binary tree and the generalization of predicates is based on the generalizations of pairs of corresponding terms in the predicates. This program also provide dynamic update on generalization in the presence of a binding. The following is an example of how a generalization of predicates is made.
Finding the generalization for $p(f(x),y), p(z,w), p(f(w),q)$ and $p(f(z),r)$:

$$
\begin{array}{ll}
p(f(x),y) & v0 = \Phi(f(x),z) \\
p(z,w) & v1 = \Phi(y,w) \\
\ldots\ldots\ldots\ldots & \\
p(v0,v1) & \\
 & \\
p(f(w),q) & v2 = \Phi(w,z) \\
p(f(z),r) & f(v2) = \Phi(f(w),f(z)) \\
\ldots\ldots\ldots\ldots & v3 = \Phi(q,r) \\
p(f(v2),v3) & \\
 & \\
p(v0,v1) & v4 = \Phi(v0,f(v2)) \\
p(f(v2),v3) & v5 = \Phi(v1,v3) \\
\ldots\ldots\ldots\ldots & \\
p(v4,v5) &
\end{array}
$$

First, $p(v0,v1)$ is generated as the generalization of $p(f(x),y)$ and $p(z,w)$ in the way such that $v0$ is obtained from the generalization of $f(x)$ and z and $v1$ is obtained from the generalization of y and w. Next, $p(f(v2),v3)$ is generated as the generalization of $p(f(w),q)$ and $p(f(z),r))$ such that $f(v2)$ is the generalization of $f(w)$ and $f(z)$ and $v3$ is the generalization of q and r. Finally, $p(v4,v5)$, the generalization of the four predicates is generated from $p(v0,v1)$ and $p(f(v2),v3)$.

Now suppose that z is bound to $f(z1)$, the generalization of $f(x)$ and z should be changed from $v0$ to $f(v0')$. $v0$ is bound to $f(v0')$ and this invoke the reconsideration of the generalization of $v0$ and $f(v2)$. The generalization of these two terms becomes $f(v4')$ and $v4$ should be bound to $f(v4')$.

The general predicate $p(v0,v1)$ becomes $p(f(v0'),v1)$ and the final generalization $p(v4,v5)$ becomes $p(f(v4'),v5)$ after the binding of z. In this application, neither of these predicates will be affected directly because we could always get the new term through the old term. For example, to obtain the current generalization of the four predicates, $p(f(v4'),v5)$, all we have to do here is to follow index from $v4$ to $f(v4')$. No generalization of predicates will be affected once it is generated.

The basic data structures used in this program include a symbol table and a hash table for terms, a predicate table for predicates and a general predicate table for

general predicates which are generated from predicates. Notice that the index to a term means the position of the term in the symbol table.

- symtable: a symbol table such that each element contains the following.
  1. lexptr: the name of the term.
  2. type: the type of the term, variable or functor.
  3. fp: includes indices to the subterms of the term.
  4. nextp: index to the term it is bound to.
  5. arg: number of subterms in the term.
  6. tolist: an array contains indices to elements in genTable, which are generalizations could be affected by the binding of this term. When this term is bound to another term, this list is followed and all the generalizations pointed to by the indices should be reconsidered, thus, the list provides direct accessing for generalization update.
- ht: a hash table for supporting symbol table. Each element contains:
  1. list: an array contains indices to the terms in the symbol table.
- pred: a table contains predicates such that each element contains the following:
  1. lexptr: the name of the predicate.
  2. fp: includes indices to the terms in the predicate.
  3. arg: number of terms in the predicate.
  4. g: index to the general predicate relates to this predicate. In the first example, the g of $p(f(x), y)$ is the index of $p(v0, v1)$ in the preg table.
- preg: a table contains general predicates. Each element contains:
  1. lexptr: the name of the predicate.
  2. fp: includes indices to the terms in the general predicate.
  3. arg: number of terms in the predicate.
  4. g: index to the general predicate relates to this general predicate. In the first example, the g of $p(v0, v1)$ is the index of $p(v4, v5)$ in the preg table.
- Gen: a two dimensional table that records all the generalizations. If the least general term of the ith and the jth terms in the symbol table is the kth term in the symbol table then Gen[i][j] = k. If there is no least general term generated for the ith term and the jth term then Gen[i][j] = 0. Whenever a generalization is generated, a proper element in this table will be updated.
- genTable: a table contains generalizations which may need to be updated when bindings occur. Each element contains:

1. one: index to the first term of the generalization.
2. two: index to the second term of the generalization.
3. gen: index to the general term of the first and the second terms.

Whenever a generalization with variable as generalization is generated, a proper element will be added to this table.

- queue: a queue contains new bindings which are generated during term update. Each element contains (old, new) such that the (old) term should be bound to the (new) term.

The program is organized in the following files:

- global.h contains the definitions of the global data structures.
- lex.c contains procedures for data initialization and procedures which receive input.
- symbol.c contains procedures that manage the symbol table, hash table and predicate tables; procedures for insertion and searching.
- gene.c contains procedures for generalization.
- change.c contains procedures for updating in the presence of a binding. New bindings which are generated during update are also taken care of automatically by these procedures.

The following are part of the table contents after the generalization of $p(f(x,y),y)$ and $p(f(z,w),z)$.

$p(f(x,y),y)$
$p(f(z,w),z)$
.....
$p(f(v1,v2),v3)$

pred: the predicate table

	lexptr	$fp(terms)$	
1	p	$1,3$	$(p(f(x,y),y))$
2	p	$4,5$	$(p(f(z,w),z))$

preg: the general predicate table

	lexptr	$fp(terms)$	
1	p	$9,10$	$(p(f(v1,v2),v3))$

symtable: the symbol table

	lexptr	*type*	*fp(subterms)*	*tolist*	*nextp*	*(the whole term)*
1	f	*functor*	$2,3$			$(f(x,y))$
2	x	*variable*		$<1>$		
3	y	*variable*		$<2,3>$		
4	f	*functor*	$5,6$			$(f(z,w))$
5	z	*variable*		$<1,3>$		
6	w	*variable*		$<2>$		
7	$v1$	*variable*				
8	$v2$	*variable*				
9	f	*functor*	$7,8$			$(f(v1,v2))$
10	$v3$	*variable*				

ht: the hash table

	Index	*to*
1	(2)	(x)
2	(3)	(y)
3	(5)	(z)
4	(6)	(w)
5	(1)	$(f(x,y))$
6	(4)	$(f(z,w))$
7	(7)	$(v1)$
8	(8)	$(v2)$
9	(10)	$(v3)$
10	(9)	$(f(v1,v2))$

Gen: the generalization table

Gen[2][5] = 7; v1 = Φ(x, z).
Gen[3][6] = 8; v2 = Φ(y, w).
Gen[1][4] = 9; f(v1, v2) = Φ(f(x, y), f(z, w)).
Gen[3][5] = 10; v3 = Φ(y, z).

genTable: table for generalizations which may need to be updated.

	$< one, two, gen >$	
1	$<2,5,7>$	$v1 = \Phi(x,z).$
2	$<3,6,8>$	$v2 = \Phi(y,w).$
3	$<3,5,10>$	$v3 = \Phi(y,z).$

The hash table supports term searching; for example, the hash function locates $f(x,y)$ to the the fifth element of the hash table, which contains index 1 to the symbol

table, thus the first term is compared with $f(x,y)$, which takes time proportional to the size of the term to make sure it is the $f(x,y)$ we are looking for. Without hash table, we may have to compare all the terms in the symbol table with the term we are searching for.

Also notice the relationship between the tolist in each element of the symbol table and the elements in genTable; The tolist of z as we could see contains indices, 1 and 3, to the elements in the genTable because $v1$ and $v3$ may need to be updated when z is bound to another term. The use of tolist makes it possible to access the generalizations directly without checking all the generalizations. To obtain the tolist of a term, an index should be inserted each time a reconsiderable generalization which relates to this term is generated.

Suppose that x is bound to y, the nextp of x will be 3 (the index to y), and the index in the tolist of x, 1, leads us to the first element of genTable, thus, the generalization of x and z, which becomes the generalization of y and z, should be reconsidered. But from Gen[3][5] = 10, we find out the generalization of y and z already exists, therefore, instead of generating new generalization, $v1$ should be bound to $v3$.

We should also be aware of the fact that when a binding occurs, the index of an element in the genTable sometimes needs to be added to the tolists of other terms. In the example above, if x is bound to $f(q,r)$, the first element of the genTable is reconsidered. Although the generalization, $v1$, remains the same, the index of this element should be added to the tolists of q and r because the generalization may be affected when the binding of q or r occurs.

The update of generalization will not directly affect predicates and general predicates because the contents in both pred and preg table are referred through symbol table. The new status of a predicate could be obtained through the nextp indices of the terms in the predicate. No individual effort is needed to maintain the contents in the predicate and general predicate tables.

The following pages are the codes of the program, which also include some descriptions for more details.

References

[1] J-L. Lassez, M.J. Maher and K.Marriott. Unification Revisited. In *Foundations of Deductive Databases and Logic Programming,* Morgan Kaufmann Publishers, 1988.

[2] C. David Page Jr., Alan M. Frisch. Generalization with Taxonomic Information. In *Proceedings of AAAI-90,* Boston, MA, 1990.

[3] G. D. Plotkin. A Note on Inductive Generalization, chapter 8, Volumn 5 of *Machine Intelligence,* Edinburgh University Press, 1970.

[4] G. D. Plotkin. A Further Note on Inductive Generalization, chapter 8, Volumn 6 of *Machine Intelligence,* Edinburgh University Press, 1971.

[5] J. C. Reynolds. Transformational Systems and Algebraic Structure of Atomic Formulas, chapter 7, Volumn 5 of *Machine Intelligence,* Edinburgh University Press, 1970.

[6] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define BSIZE 128
#define EOS '\0'
#define V 1
#define F 2

#define HSIZE 197
#define STRMAX 999
#define SYMMAX 100
#define FALSE 0
#define TRUE 1
#define MaxA 20

extern int  tokenval;
extern int  tempv;
extern int  countg;
extern int  teminx[];
extern int  tempinx[];

extern char lexemes[];
extern int  lastchar;
extern int  lastgenT;
extern int  lastentry;
extern int  lastpre;
extern int  lastpg;
extern int  oldlastpre;

struct gTable {
  int one;
  int two;
  int gen;
};

struct tlist {
  int gen;
  int updw;

};

struct pgtable {
  char *lexptr;
  int g;
  int arg;
  int nump;
  int fp[BSIZE];


};
struct entry {
  char *lexptr;
  char *lexptr1;
  int g;
  int gnum;
  int arg;
  int type;
  int nextp;
  int gpos;
  int pos;
  int place;
  int ingen;
  struct tlist tolist[20];
  int fp[BSIZE];
```

```
};


struct ht2 {
  char *name;
  int entry;
};

struct ht1 {

        int num;
        struct ht2 list[20];
};


struct ht1 ht[HSIZE];


struct gTable  genTable[SYMMAX];
struct entry   symtable[SYMMAX];
struct entry   pred[SYMMAX];
struct pgtable preg[SYMMAX];
int Gen[SYMMAX][SYMMAX];
int inxt2[SYMMAX][SYMMAX];
int inxt3[SYMMAX][SYMMAX];/* started from 1, 0 is the number*/
int Contain[SYMMAX][SYMMAX];
extern int  checkT();
extern int  hacon();
extern int  insert();
extern int  insertp();
extern void error();
extern int  getpre();
extern int  getfname();
extern int  check();
extern int  getf2();
extern int  getarg();
extern int  findfun();
extern int  gen();
extern int  same();
extern int  chf();
extern void  printgenp();
extern char  *newT();
```

```
/*****************************************************************/
/*    lex.c:                                                     */
/*    This file contains the main procedure and the procedures   */
/*    which receives predicates and terms from input.            */
/*    The procedures includes:                                   */
/*    main: the main procedure.                                  */
/*    init                                                       */
/*    lexs                                                       */
/*    chV                                                        */
/*    printprep                                                  */
/*    printgenp                                                  */
/*    printsym                                                   */
/*    getpre                                                     */
/*    getfname                                                   */
/*    getf2                                                      */
/*    getarg                                                     */
/*    findfun                                                    */
/*****************************************************************/
#include  "global.h"
#include <sys/types.h>

char  lexbuf[BSIZE];
int tokenval = -1;
int exist = 0;
int oldlastpre;
void lexs();
void chV();
int getpre();
int getfname();
int check();
int getf2();
int getarg();
int findfun();
void printprep();
void printgenp();
void printsym();
void init();

main()
{
  char n ;
  int i,j,k;

  init();                          /*initialize data structures*/

  do {
    printf("\n\n1: Add/more predicates.\n");
    printf("2: Bind a variable.\n");
    printf("3: Quit.\n");
    printf("Your choice? \n");
    scanf("%1c",&n);
    printf("\n");

    if (n == '1')
      lexs();                      /*receive input and generalization*/
    else if (n == '2')
      chV();                       /*bind a term to another*/

    else if (n == '3')
      break;
  } while (n != '3');
}

/*****************************************
  init:
```

```
  called by main.
  Initializes data structure.
 ******************************************/
void init()
{
  int i,j,k;
  for (i = 0 ;i<HSIZE;i++)
    ht[i].num = 0;

  for (i = 0; i<SYMMAX;i++)
    {
      for (k =0;k<BSIZE;k++)
        {
          preg[i].fp[k] = 0;

        }
      preg[i].nump = 0;
    }
 for (i =0;i<SYMMAX;i++)
    {
      for (j = 0;j<BSIZE;j++)
        {
          symtable[i].fp[j] = 0;
          pred[i].fp[j] = 0;
          preg[i].fp[j] = 0;
        }
      pred[i].g = 0;
      preg[i].g = 0;
      symtable[i].ingen = 0;
      symtable[i].nextp =0;

    }
}

/*****************************************
  lexs:
  called by main.
  calls getpre, gen, printprep,
       printgenp and printsys.
  receives input and generalizes.
 *****************************************/
void lexs()
{
  int t=0,i=1,j=0,k=0;

  oldlastpre = lastpre;

  /*get predicates from input*/
  while((t = getchar()) !='*')
    {
      ungetc(t,stdin);
      getpre();
    }

  printprep();
  /* Generalize predicate*/
  gen();
  /* Output the status of generalization */
  printgenp();
  printsym();
}

/*****************************************
  chV:
  called by main;
```

```
  calls chf, printprep,
       printgenp and printsys.
  prestep for term bindings.
 ******************************************/
void chV()
{
  char s[128];
  char t[128];


  printf("Which variable you want to bound? ");
  scanf("%s",s);
  printf("Bind to ? ");
  scanf("%s",t);
  /* Bind s to t*/
  chf(t,s);
  /* Output the status after binding */
  printprep();
  printgenp();
  printsym();
}

/*****************************************
  printprep:
  called by lexs and chV.
  Outputs predicates.
 *****************************************/
void printprep()
{
  int i,j;

  printf("\n");

  for (i=1;i<=lastpre;i++)
    {
      printf("\n%d:  %s(",i,pred[i].lexptr);
      for (j=1;j<=pred[i].arg;j++)
        {

          printf("%s",symtable[pred[i].fp[j]].lexptr);
          if (j != pred[i].arg )
            printf(",");
        }
      printf(")\n");

    }
  printf("----------------------------\n");
}

/*****************************************
  printgenp:
  called by lexs and chV.
  Outputs general predicates.
 *****************************************/
void printgenp()
{
  int i,j,k;

  printf("\n");

  for ( i = 1;i< lastpg; i++)
    {
      printf("\n%d:  %s(",i,preg[i].lexptr);
      for (j=1;j<=preg[i].arg;j++)
        {
```

```
            printf("%s",symtable[preg[i].fp[j]].lexptr);
            if (j != preg[i].arg)
              printf(",");
          }
        printf(")\n");

      }
    printf("----------------------------\n");
}

/*****************************************
  printsym:
  called by lexs and chV.
  Outputs Terms in symbol table.
 *****************************************/
void printsym()
{
  int i,j,next,k;

  printf("\n");

  for ( i = 1; i<=lastentry;i++)
    {printf(" %d :  %s ",i,symtable[i].lexptr);
     j=0;

     k = i;
     while (symtable[k].nextp !=0 && symtable[k].nextp !=i)
       {
         k = symtable[k].nextp;
       }
     /*k = symtable[i].nextp;*/
     if ( k != i && symtable[k].lexptr !=NULL)
       printf(" --> %d :  %s ",k,symtable[k].lexptr);

     printf("\n");
    }


}

static int num =0;

/*****************************************
  getpre:
  called by lexs.
  calls getfname and getf2.
  gets a predicate from input.
 *****************************************/
int getpre()
{
  int t,p;
  int j=0;
  exist = 0;

  /* get the predicate name */
  p = getfname();

  t = getchar();
  while(t == ' ' ||t == '\t')
    t = getchar();

  /* Get the terms in the predicate */
  if (t == '(')
    getf2(p);
```

```
}

/*****************************************
  getfname:
  called by getpre.
  calls insertp.
  gets the predicate name from input.
 *****************************************/
int getfname()
{
  int t,p;
  t = getchar();
  while(!isalpha(t))
    t = getchar();

      if (isalpha(t))
        {
          int p,b = 0;
          while (isalnum(t))
            {
              lexbuf[b] = t;
              t = getchar();
              b = b+1;
              if (b >= BSIZE)
                error("compiler error");
            }
          lexbuf[b] = EOS;
          /*if (t != EOF)*/
            ungetc(t,stdin);

          /* insert the predicate name to predicate table and return a index*/
          p=insertp(lexbuf);
          return p;
        }
      else if (t == EOF)
        return 0;

}

/*****************************************
  getf2:
  called by getpre, getarg and findfun.
  calls getarg.
  Gets the terms in the predicate from input.
 *****************************************/
int getf2(p)
{
  int t,i=0,j = 0;
  int r = 1;


  if (exist ==1)
    {
      /* there exists predicate with the same name,
         the number of terms inside should be the same .*/

      for (i = 0,j = 1 ;i<pred[p].arg;i++,j++)
        {
          /* get a term from input and return a index to the term*/
          r=getarg(p);
          if (r !=0)
            pred[p].fp[j] = r;

        }
```

```
    }
  else
    {

      num =0;
      i =1;
      while (r != 0)
        {
          /* get a term from input and return a index to the term*/
          r = getarg(p);
          if ( r != 0)
            {
              pred[p].fp[i] = r;
              i++;
            }
        }
      pred[p].arg = i-1;

    }
}

/*****************************************
  getarg:
  called by genf2.
  calls checkT, getf2 and insert.
  gets a term from input fro predicate,
  inserts the term to symbol table
  and returns a index to the term.
 *****************************************/
int getarg(l)
int l;
{
  int t,i;
  int type;
  int count = 0;
  type = V;
  num++;
  while(1)
    {
      t = getchar();

      if (t == ' ' ||t == '\t' || t == '\n')
        ;
      else if (t == ')' )
        return 0;
      else if (isdigit(t))
        {
          ungetc(t,stdin);
          scanf("%d",&tokenval);
          return 0;
        }
      else if (isalpha(t))
        {
          int p,q,b = 0;
          while (isalnum(t) || ( t == ',' && count >0) ||
                 t == '(' || t == ')' && count>0)
            {
              if ( t == '(')
                {
                  /* the term is a functor */
                  count++;
                  type = F;
                }
              else if (t == ')')
                count--;
```

```
            lexbuf[b] = t;
            t = getchar();
            b = b+1;
            if (b >= BSIZE)
              error("compiler error");
          }
        lexbuf[b] = EOS;

        if (t != EOF && t != ',')
          ungetc(t,stdin);
        if (t == '(')
          {
            p = checkT(lexbuf);
            getf2(p);
            return p;
          }
        /* check if the term exists */
        p = checkT(lexbuf);


        if (p == 0)
          {
            /* the term does not exist,
               it should be inserted to symbol table*/
            exist = 0;
            p = insert(lexbuf,type);

          }
        else if (q == 0)
            exist = 1;
        return p;
      }
    else
      return 0;

  }
}

/*******************************************
  findfun:
  called by insert.
  calls checkT, getf2 and insert.
  Inserts the terms inside a term
  to the symbol table
 *******************************************/
int findfun(j,s)
int j;
char s[];
{

  int t,p;
  int i = 0,k=0;

  int count;
  int type;
  type =V;

  symtable[j].arg = 0;

  /* skip through the function name */
  while (s[i] != '(')
    i++;
  i++;
```

```
/* Get a term inside the term and insert it to symbol table */
while(s[i] != '\0')
  {
    type = V;
    count = 0;
    if (s[i] == ' ' ||s[i] == '\t' || (s[i] == ',' && count == 0)
        || s[i] == '\n')
      ;
    else if (s[i] == ')')
      return 0;
    else if (isdigit(s[i]))
      {
        ungetc(s[i],stdin);
        scanf("%d",&tokenval);
        return 0;
      }
    else if (isalpha(s[i]))
      {
        int p,b = 0;
        while (isalnum(s[i]) || (s[i] == ',' && count > 0)
               || s[i] == '(' || s[i] == ')' && count > 0)
          {
            if ( s[i] == '(')
              {
                count++;
                type =F;
                lexbuf[b] = '\0';
              }
            else if (s[i] == ')')
              count--;

            lexbuf[b] = s[i];

            b = b+1;
            if (b >= BSIZE)
              error("compiler error");
            i++;
          }
        lexbuf[b] = EOS;
        if (s[i] == '(')
          {
            p = checkT(lexbuf);
            getf2(p);
            return p;
          }

        /* check if the term exists */
        p = checkT(lexbuf);
        if (p == 0)
          {
            /* the term does not exist,
               it should be inserted to symbol table*/
            exist = 0;
            p = insert(lexbuf,type);
          }
        else
          exist = 1;
        symtable[j].fp[k] = p;
        symtable[j].arg++;
        k++;

      }
    i++;
  }
```

}

```
/*****************************************************************/
/*    symbol.c:                                                  */
/*    contains procedures that manage the symbol table,          */
/*    hash table and predicate table; procedures for insertion   */
/*    and searching.                                             */
/*    The procedures includes:                                   */
/*    checkT                                                     */
/*    insert                                                     */
/*    hashing                                                    */
/*    insertp                                                    */
/*    hacon                                                      */
/*****************************************************************/
#include "global.h"
#include <stdlib.h>
char lexemes[STRMAX];
int  lastchar = -1;

int  lastentry =0;
int  lastpre =0;
int tempv =0;
int teminx[20];
int hashing();
int hacon();

/*****************************************
  checkT:
  called by getarg, chf, ch1, decide,
  newT and makeV.
  calls hacon.
  Checks the hash table if a term exist
 *****************************************/
int checkT(s)
char s[];
{
  int i;
  int va,n;
  int ha;

  ha = 0;
  va = hacon(s); /* obtain a hash value*/

  n = ht[va].num;

  /* compare the terms in a position of hash table with the term we request*/
  for (i = 1; i<=n; i++)
    {
      if (strcmp(s,ht[va].list[i].name) == 0)
        {

          ha = ht[va].list[i].entry;
          return ha;
        }
    }
  return ha;
}

/*****************************************
  insert:
  called by getarg, makenewF and genF.
  calls hashing, hacon and findfun.
  inserts a term to both symbol and hash table.
 *****************************************/

int insert(s,type)
     char s[];
```

```
    int type;
{
  int p,pos;
  int  ha;

  if (lastentry +1 >= SYMMAX)
    error("symbol table full");

  /* insert the term to the end of the symbol table*/
  lastentry = lastentry+1;
  symtable[lastentry].type = type;
  p = lastentry;

  /* insert the term into hash table with insex to symbol table*/
  ha = hashing(s,p);
  pos = hacon(s);
  /*symtable[p].pos = pos;
  symtable[p].place = ha;*/

  /* if it is a functor,
     the terms inside should also be inserted*/
  if (type ==F)
    findfun(lastentry,s);


  if (symtable[p].lexptr == NULL)
    symtable[p].lexptr = symtable[p].lexptr1 = ht[pos].list[ha].name;
  else
    ht[pos].list[ha].name = symtable[p].lexptr;
  return p ;
}

/*****************************************
  hashing:
  called by insert, newT.
  calls hacon.
  inserts a term s to hash table
  with index p to symbol table.
 *****************************************/
int hashing(s,p)
char s[];
int p;
{
  int va,n,len;
  int ha;

  /* Obtain a hash value */
  va = hacon(s);
  ht[va].num++;
  n = ht[va].num;

  /* insert the term into the hash table */
  ht[va].list[n].name= &lexemes[lastchar +1];
  strcpy(ht[va].list[n].name, s);
  len = strlen(s);
  lastchar = lastchar + len +1;

  /* with index p to a entry in the symbol table */
  ht[va].list[n].entry = p;

  symtable[p].lexptr1 = ht[va].list[n].name;
  symtable[p].pos = va;
  symtable[p].place = ha;
  ha = n;
```

```
  return ha;
}

/*****************************************
  insertp:
  called by getfname.
  inserts a predicate s to predicate table,
  but only predicate name at this moment.
 *****************************************/
int insertp(s)
     char s[];

{
  int len;
  len = strlen(s);

  if (lastpre +1 >= SYMMAX)
    error("symbol table full");

  /* insert to the end of the table*/
  lastpre = lastpre+1;
  pred[lastpre].lexptr = &lexemes[lastchar +1];
  pred[lastpre].g =0;

  lastchar = lastchar + len +1;
  strcpy(pred[lastpre].lexptr,s);
  return lastpre;
}

/*****************************************
  hacon:
  called by checkT, insert, hashing and newT.
  generates a hash value for term s.
 *****************************************/
int hacon(s)
char s[];
{
  long val;
  int va;

  val = atol(s);
  va = val % HSIZE;
  return va;
}
```

```
/*****************************************************************/
/*    gene.c:                                                    */
/*    contains procedures for generalization.                    */
/*    The procedures includes:                                   */
/*    gen                                                        */
/*    genre                                                      */
/*    existg                                                     */
/*    same                                                       */
/*    Hvalue                                                     */
/*    makeV                                                      */
/*    genT                                                       */
/*    listg                                                      */
/*    sameF                                                      */
/*    genF                                                       */
/*    itoa                                                       */
/*    reverse                                                    */
/*    putinx                                                     */
/*****************************************************************/
#include "global.h"
#include <stdlib.h>
int tempinx[20];
int lastpg=1;
int lastgenT=0;
int countg = 1;
void itoa();
int existg();
int genre();
int putinx();

/*************************************************************
  gen:
  Finds the generalization of several predicates.
 *************************************************************/
int gen()
{
  int i,j=0,k,l,m,p;
  int len;
  int ge,g;
  int oldnum;

  for (i =0; i<20; i++)
    teminx[i] = 0;
  for (i =0; i<20; i++)
    tempinx[i] = 0;

  /* exam only those newly inserted predicates */
  for (i =1+oldlastpre;i<=lastpre;i++)
    {
      j =0;

      if (pred[i].g == 0)
        {
          /* This predicate does not have generalization yet,
             find those new predicates with the same predicate name
             and generate a generalization for them */

          ge = existg(pred[i].lexptr);
          tempinx[j] = i;
          j++;

          /* Get the predicates with the same predicate name*/
          for (k =i+1;k<=lastpre;k++)
            if (strcmp(pred[i].lexptr,pred[k].lexptr) ==0)
              {
                tempinx[j] = k;
```

```
                j++;
              }
            /* genre will find the generalization for these j new predicates*/
            g = genre(0,j-1);

            /* g is the generalization for these j predicates.
               if there is a old generalization with the same predicate name,
               generate the generalization for the old one and g */

            if (ge != 0)
              {
                tempinx[0] = ge;
                tempinx[1] = g;
                g = genre(-1,1);
              }
          }
      }
  }

  /*************************************************************
    existg:
    called by gen;
    checks if there is a generalization for predicate with
    predicate name s exists.
   *************************************************************/
  int existg(s)
  char s[];
  {
    int i;

    for (i = 1;i<lastpg;i++)
      if (strcmp(s,preg[i].lexptr) == 0)
        {
          while (preg[i].g != 0)
            i = preg[i].g;

          return i;
        }

    return 0;
  }

  /*************************************************************
    same:
    called by genF,genre.
    calls putinx, Hvalue, genF and decide.
    finds the least general term for terms p and q, the
    general term   will be inside general predicate l.
   *************************************************************/
  int same(p,q,l)
  int p;
  int q;
  int l;

  {
    int r=0;
    int i,j,k,g;

    if (p != q)
      {
        r = 0;
      }
    else
      r = 2;
```

```
  /* p anq q are the same term,
     the generalization is itself*/
  if (r ==2)
    {
      putinx(l,p);
      return p;
    }

  /* if the generalization has already exist,
     return the index to the general term*/
  if ((g = Gen[p][q])!=0)
    return g;

  if (symtable[p].type == F && symtable[q].type == F)
    r = 1;
  else
    r = 0;


  if (r == 0)
    {
      /* At least one of the two terms is a variable,
         generate a variable term as their generalization*/
      k = Hvalue(p,q,l);
      return k;
    }
  else
    {
      k = sameF(p,q);

      if ( k !=  0)
        {
          /* The two terms are functors with different names,
             generate a variable term as their generalization*/
          k = Hvalue(p,q,l);
          return k;
        }
      else /* The two terms are functors of the same name,
              generate a function term as their generalization*/
        k= genF(p,q,l);
      return k;
    }

}

/*************************************************************
  Hvalue:
  called by same;
  calls makeV.
  generates a variable term as the generalization of k and h.
  the generalization will be inside general predicate l
 *************************************************************/
int Hvalue(k,h,l)
int k;
int h;
int l;
{

  int n,i,gn;
  int q =0;

  /*generate a variable term
    and insert it to inxt table*/
  n = makeV(k,h,l);
  Gen[k][h] = n;
```

```
  gn = symtable[n].gnum;
  /*this generalization is inside predicate l*/
  inxt2[gn][l] = 1;

  return n;

}

/*************************************************************
  makeV:
  called by Hvalue;
  calls checks, genT and listg.
  generates a new variable as the generalization of k and h
 *************************************************************/
int makeV(k,h,l)
int k,h,l;
{
  char s[128];
  char ss[128];
  int len;
  int tempnum,p;
  int pos;
  int gnum,num;

  tempnum = tempv+ atoi("0");
  itoa(tempnum,s);
  strcpy(ss,"temp");
  strcat(ss,s);

  if ((p = checkT(ss)) !=0)
    {
      gnum = symtable[p].gnum;
      num = inxt3[gnum][0];
      inxt3[gnum][num] = l;
      inxt3[gnum][0] = inxt3[gnum][0]+1;
      tempv++;
      return p;
    }


  if (lastentry +1 >= SYMMAX)
    error("symbol table full");
  lastentry = lastentry+1;
  symtable[lastentry].lexptr = symtable[lastentry].lexptr1 =&lexemes[lastchar +1];
  symtable[lastentry].gnum = countg++;
  symtable[lastentry].type = V;

  tempnum = tempv+ atoi("0");
  itoa(tempnum,s);

  strcpy(symtable[lastentry].lexptr,"temp");
  strcat(symtable[lastentry].lexptr,s);
  len = strlen(symtable[lastentry].lexptr);
  lastchar = lastchar + len +1;
  tempv++;
  symtable[lastentry].gnum = countg++;
  hashing(symtable[lastentry].lexptr,lastentry);
  pos = genT(k,h,lastentry);
  symtable[lastentry].gpos = pos;
  listg(k,pos,1,lastentry);
  listg(h,pos,2,lastentry);
  gnum = symtable[lastentry].gnum;
  inxt3[gnum][0] = 1;
  inxt3[gnum][1] = 1;
```

```
  return lastentry;
}

/*********************************************************
  genT:
  called by makeV;
  l is the generalization of terms k and h, keeps
  this information in genTable and returns the index
  of the element.
 *********************************************************/
int genT(k,h,l)
int k;
int h;
int l;
{
  lastgenT++;
  genTable[lastgenT].one = k;
  genTable[lastgenT].two = h;
  genTable[lastgenT].gen = l;

  return(lastgenT);
}

/*********************************************************
  listg:
  called by makeV;
  adds the index p of genTable to the list of term k.
 *********************************************************/
int listg(k,p,n,e)
     int k,p,n,e;
{
  int i;
  if (symtable[k].type == V)
    listgen(k,p,n,e);
  else
    {
      for (i = 0;i<symtable[k].arg;i++)
        listg(symtable[k].fp[i],p,n,e);
    }
}

int listgen(k,p,n,e)
     int k,p,n,e;
{
  int i;
  i = symtable[k].ingen++;
  symtable[k].tolist[i].gen = p;
  symtable[k].tolist[i].updw = n;

}

/*********************************************************
  sameF:
  called fy same.
  checks if term p and q are functor with the same
  function name.
 *********************************************************/
int sameF(p,q)
int p;
int q;
{
  char *s1,*s2;
  int i = 0;

  s1 = symtable[p].lexptr;
```

```
  s2 = symtable[q].lexptr;

  while (strncmp(s1,s2,1) == 0 && strncmp(s1,"(",1) != 0)
    {
      s1++;
      s2++;
    }

  if (strncmp(s1,s2,1) == 0)
    return 0;
  else
    return -1;
}

/*************************************************************
  genF:
  called by same.
  calls findfun and insert.
  generates a function term as the generalization of two
  function terms k and h.
  the generalization will be inside general predicate l.
 *************************************************************/
int genF(p,q,l)
int p;
int q;
int l;

{

  int i,k,j=0;

  int fp[BSIZE];
  char buff[BSIZE];
  int oldchar,len;
  int entry;

  for(i =0;i<BSIZE;i++)
    fp[i] = 0;

  for (i =0;i<symtable[p].arg;i++)
    fp[i] = same(symtable[p].fp[i],symtable[q].fp[i],l);

  oldchar = lastchar;
  strcpy(buff,symtable[p].lexptr);

  i = 0;
  while (strncmp(buff+i,"(",1) !=0)
    i++;
  strcpy(buff+i+1,"\0");

  for (i =0;i<symtable[p].arg;i++)
    {
      strcat(buff,symtable[fp[i]].lexptr);
      if (i != symtable[p].arg-1)
        strcat(buff,",");

    }
  strcat(buff,")");
  entry = insert(buff,V);
  symtable[entry].arg = symtable[p].arg;
  for (i =0;i<symtable[p].arg;i++)
    symtable[entry].fp[i] = fp[i];

  Gen[p][q] = entry;
  symtable[entry].type = F;
```

```
  symtable[entry].gnum = countg++;
  inxt2[symtable[entry].gnum][1] = 1;
  findfun(entry,symtable[entry].lexptr);

  return entry;

}

/*************************************************************
  reverse:
  called by itoa;
  reverses string s.
 *************************************************************/
void reverse(s)
char s[];
{
  int c,i,j;

  for (i = 0,j = strlen(s)-1;i <j ; i++,j--)
    {
      c = s[i];
      s[i] = s[j];
      s[j] = c;
    }
}

/*************************************************************
  itoa:
  called by makeV.
  calls reverse.
  turns integer to string
 *************************************************************/
void itoa(n,s)
int n;
char s[];
{

  int i,sign;

  if ((sign = n) <0)
    n =-n;
  i = 0;
  do {
    s[i++] = n % 10 +'0';
  } while ((n /= 10) >0);

  if (sign <0)
    s[i++] = '-';

  s[i] = '\0';
  reverse(s);
}

/*************************************************************
  genre:
  called by gen.
  calls same, putinx and genre.
  generates a generalization for several predicates.
 *************************************************************/
int genre(i,j)
int i;
int j;
{
  int p;
  int q;
```

```
int g;
int l;
int len;

if (i == -1)
  {
    /*we are going to find the generalization of
      a old generalization and a newly generated generalization*/

    preg[tempinx[1]].g = lastpg;
    preg[tempinx[0]].g = lastpg;
    preg[lastpg].arg = preg[tempinx[1]].arg;
    preg[lastpg].nump = 2;

    /*find the least generalization for each pair of terms
      in the two predicate*/
    for (l = 1;l<=preg[tempinx[1]].arg;l++)
      {
        teminx[0] = preg[tempinx[0]].fp[l];
        teminx[1] = preg[tempinx[1]].fp[l];
        g = same(teminx[0],teminx[1],lastpg);
        preg[lastpg].fp[l] = g;
      }
    preg[lastpg].lexptr = &lexemes[lastchar +1];
    strcpy(preg[lastpg].lexptr,preg[tempinx[1]].lexptr);

  }
else if (j-i>=2)
  {
    /* there are more than two predicates,
       a divide and conquer approach is used here*/
    p = genre(i,(int)(j-i)/2+i);
    q = genre((int)(j-i)/2+i+1,j);

    /* p and q are two general predicates obtained from two groups
       of predicates, we are going to generate the generalization of p and q*/
    preg[p].g = lastpg;
    preg[q].g = lastpg;
    preg[lastpg].arg = preg[p].arg;
    preg[lastpg].nump = 2;

    /*find the least generalization for each pair of terms
      in the two predicate*/
    for (l = 1;l<=pred[p].arg;l++)
      {
        teminx[0] = preg[p].fp[l];
        teminx[1] = preg[q].fp[l];
        g = same(teminx[0],teminx[1],lastpg);
        preg[lastpg].fp[l] = g;
      }
    preg[lastpg].lexptr = &lexemes[lastchar +1];
    strcpy(preg[lastpg].lexptr,preg[p].lexptr);
  }
else if (j-i<1)
  {
    /* there is only one predicate, the generalization
       will be obtained based on the predicate itself */
    pred[tempinx[j]].g = lastpg;
    preg[lastpg].arg = pred[tempinx[j]].arg;
    preg[lastpg].nump = 1;
    for (l = 1;l<=pred[tempinx[j]].arg;l++)
      {
        preg[lastpg].fp[l] = pred[tempinx[j]].fp[l];
        putinx(lastpg,pred[tempinx[j]].fp[l]);
      }
```

```
        preg[lastpg].lexptr = &lexemes[lastchar +1];
        strcpy(preg[lastpg].lexptr,pred[tempinx[j]].lexptr);
      }
    else
      {
        /* there are exactly two predicates*/

        pred[tempinx[j]].g = lastpg;
        pred[tempinx[i]].g = lastpg;
        preg[lastpg].arg = pred[tempinx[j]].arg;
        preg[lastpg].nump = 2;

        /*find the least generalization for each pair of terms
          in the two predicate*/
        for (l = 1;l<=pred[tempinx[j]].arg;l++)
          {
            teminx[0] = pred[tempinx[i]].fp[l];
            teminx[1] = pred[tempinx[j]].fp[l];
            g = same(teminx[0],teminx[1],lastpg);
            preg[lastpg].fp[l] = g;
          }
        preg[lastpg].lexptr = &lexemes[lastchar +1];
        strcpy(preg[lastpg].lexptr,pred[tempinx[j]].lexptr);
      }

    len = strlen(preg[lastpg].lexptr);
    lastchar = lastchar+len+1;
    lastpg++;
    return(lastpg-1);
}

int putinx(l,p)
int l;
int p;
{

    if (symtable[p].gnum== 0)
      symtable[p].gnum = countg++;

    inxt2[symtable[p].gnum][l] = 1;
}
```

```
/*****************************************************************/
/*    change.c:                                                  */
/*    contains procedures for updating in the presence of a      */
/*    binding.                                                   */
/*    The procedures includes:                                   */
/*    chf                                                        */
/*    ch1                                                        */
/*    makenewF                                                   */
/*    update0                                                    */
/*    update2                                                    */
/*    decide                                                     */
/*    cpix                                                       */
/*    newT                                                       */
/*    push                                                       */
/*    pop                                                        */
/*****************************************************************/
#include "global.h"

#define MAXVAL 100
struct rec {
  int gen;
  int arg;
  int old;
  int new;
  int nt;
  int ft;
};

struct rec queue[MAXVAL];
int newt;
int sp = 0;
int ext=0;
void push();
struct rec pop();


/*************************************************************
  decide:
  Called by update0.
  calls checkT, newT, levelp, same, cpix and push.
  decides how a generalization should be updated and
  applies with different approach.
 *************************************************************/
int decide(en,n,j,l,nt,ft)
int en;
int n;
int j;
int l;
int nt;
int ft;

{

  int g,ogen,num,ngen,oinx,f,s,of,os,bro;
  int p,i,k,old;
  int lv = 0;
  int one,two,gen;
  char buff[BSIZE];

  /* get the two terms and their generalization
     from the element in the genTable */
  one = genTable[en].one;
  two = genTable[en].two;
  gen = genTable[en].gen;
```

```
/* follow indices to the terms they have been bound to*/
while (symtable[one].nextp != 0)
  one = symtable[one].nextp;
while (symtable[two].nextp != 0)
  two = symtable[two].nextp;
while (symtable[gen].nextp != 0)
  gen = symtable[gen].nextp;

of = one;
os = two;

/* get the new status of the two terms*/
if ( one != j && one != 1)
  {
    strcpy(buff,newT(one,j,1));
    of = one;
    f = checkT(buff);
  }
else
  f = 1;

if ( two != j &&  two != 1)
  {
    strcpy(buff,newT(two,j,1));
    s = checkT(buff);
  }
 else
   s = 1;


ogen = Gen[of][os];
oinx = symtable[ogen].gnum;
g = inxt3[oinx][1];

if (one == two)
   Gen[f][s] = f;
else if (ext ==TRUE || Gen[f][s] == 0 )
  {
    /*the new generalization does not exist*/
    if (ft == FALSE || (lv = levelp(f,s,of,os)) == 0)
      {
        /* the generalization will remain a variable term*/
        if (f == s)
          {
            p = same(f,s,g);
            ngen = symtable[p].gnum;
            cpix(ngen,oinx);
            push(g,n,gen,p,newt,lv);
          }
        else
          Gen[f][s] = Gen[of][os];

      }
    else
      {
        /* the new generalization will become a functor */
        if ((f == s) && (one == two) )
          Gen[f][s] = f;
        else
          {
            p = same(f,s,g);
            ngen = symtable[p].gnum;
            cpix(ngen,oinx);
            push(g,n,gen,p,newt,lv);
          }
```

```
        }
      }
    else if ((k = Gen[f][s]) !=0)
      {
        /*the new generalization exists*/
        int h,cou, count;
        h = pred[k].gnum;

        for (cou = 1;cou <= inxt3[oinx][0];cou++)
          if (inxt2[h][inxt3[oinx][cou]] == 1)
            count++;

        if (count == 0 || gen == k)
          {
            /* the old generalization does not appear in the
               same predicate as the new generalization */
            if (ft == FALSE || (lv = levelp(f,s,of,os)) == 0)
              {
                if (gen != k)
                  symtable[gen].nextp = k;
              }
            else
              {
                teminx[0] = f;
                teminx[1] = s;
                p = same(f,s,g);
                ngen = symtable[p].gnum;
                cpix(ngen,oinx);
                push(g,n,gen,p,newt,lv);
              }
          }
        else
          {
            /* the old generalization appears in the same predicate
               as the new generalization */
            lv = levelp(f,s,of,os);
            teminx[0] = f;
            teminx[1] = s;
            p = same(f,s,g);
            ngen = symtable[p].gnum;
            cpix(ngen,oinx);
            push(g,n,gen,p,newt,lv);
          }
      }
}

int cpix(new,old)
int new,old;
{
  int i;
  for (i = 0;i<= inxt3[old][0];i++)
    inxt3[new][i] = inxt3[old][i];
}

/*************************************************************
  update0:
  Called by chf and update2.
  calls decide and update2.
  retrieves the generalizations related to the term being
  bound and updates each of these generalizations.
 *************************************************************/
int update0(j,l,nt,ft)
int j;
int l;
int nt;
```

```
int ft;

{
  int i,m,n,kind,new,h;
  int en,one,two,gen,newgen,entry;
  char buff[BSIZE];

  for (i = 0;i<symtable[j].ingen ;i++)
    {
      en = symtable[j].tolist[i].gen;
      n = symtable[j].tolist[i].updw;

      /*update the generalizations*/
      decide(en,n,j,l,nt,ft);

    }
  symtable[j].nextp = l;
  /*update2 will take care of the new bindings occur the update*/
  update2();
}


/*************************************************************
  levelp:
  called by decide.
  predict if the generalization of new status of two terms
  has more level than the generalization of the old terms
 *************************************************************/
int levelp(f,s,of,os)
int f,s,of,os;
{
  if (symtable[f].type ==F && symtable[s].type == F &&
      (symtable[of].type != symtable[f].type) ||
      symtable[os].type != symtable[s].type)
    return 1;
  else
    return 0;
}


/*************************************************************
  update2:
  called by update0.
  calls pop and decide.
  takes care of the new bindings kept in queue.
 *************************************************************/
int update2()

{
  int i,k,h,p,old;
  struct rec val1;

  while (sp>0)
    {
      val1 = pop();
      update0(val1.old,val1.new,val1.nt,val1.ft);
    }
}


/*************************************************************
  chf:
  called by chv.
  calls checkT, ch1 and update0.
  the first step of a binding, t will be bound to s.
```

```
 ************************************************************/
int chf(s,t)
char *s;
char *t;
{

  int i,j,p,k,l,m,n,ge,sib,h,kind,new;
  int ft;

  /* check whether if t exists*/
  j  = checkT(t);

  if (j !=0)
    {
      /*t exists */
      l = ch1(s,j);

      for (i =0; i<20; i++)
        teminx[i] = 0;
      if (symtable[l].type == F)
        ft = TRUE;
      else
        ft = FALSE;

      update0(j,l,newt,ft);
    }
  else
    printf("No such variable.\n");

}

/************************************************************
  newT:
  called by decide and newT.
  calls newT, checkT, hashing and hacon.
  finds the new status of p such that each occurence of q
  in p is replaced by r.
 ************************************************************/
char *newT(p,q,r)
int p;
int q;
int r;


{

  int i,j,k,len,fi = -1;
  char *l;
  char new[BSIZE];
  char buff[BSIZE];
  int fp[BSIZE];
  int pos,ha;
  int cons = TRUE;

  strcpy(buff,symtable[p].lexptr1);

  i = 0;
  while (strncmp(buff+i,"(",1) !=0)
    i++;

  strcpy(buff+i+1,"\0");

  for (i =0;i<symtable[p].arg;i++)
    {
      if (symtable[p].fp[i] == q)
```

```
        strcat(buff,symtable[r].lexptr);
      else
        {
          if (symtable[symtable[p].fp[i]].type == F )
            {
              strcpy(new, newT( symtable[p].fp[i],q,r));
              strcat(buff,new);
              if (ext == 0)
                cons = TRUE;
              ext = 0;
            }
          else
            strcat(buff,symtable[symtable[p].fp[i]].lexptr1);
        }

      if (i != symtable[p].arg-1)
        strcat(buff,",");
    }

  strcat(buff,")");
  ext = checkT(buff);
  if (ext == 0)
    ha = hashing(buff,p);

  return buff;
}

/*********************************************************
  ch1:
  called by chf.
  calls checkT, makenew.
  makes new term for s and trasfers the information from
  term j to s.
 *********************************************************/

int ch1(s,j)
char *s;
int j;
{
  int p;
  int i,k;

  p = checkT(s);

  /*check if s exists already*/
  if (p == 0)
    {
      /*insert the new term and return the index to symbol table*/
      p = makenewf(s);
      newt = TRUE;
    }
  else
    newt = FALSE;

  /* the indices to the elements in genTable which indicate generalizations
     related to the jth term should be given to the pth term(s)*/

  k = symtable[p].ingen;
  symtable[p].ingen = symtable[p].ingen + symtable[j].ingen;
  for (i = 0;i< symtable[j].ingen; i++)
    {
      symtable[p].tolist[k+i].gen = symtable[j].tolist[i].gen;
      symtable[p].tolist[k+i].updw = symtable[j].tolist[i].updw;

    }
```

```
  symtable[j].lexptr1 = symtable[p].lexptr;
  return p;
}

/*************************************************************
  makenewf:
  called by ch1.
  calls insert.
  inserts a new term s to symbol table.
 *************************************************************/
int makenewf(s)
char *s;
{
  int i,p;

  for ( i =0;i<strlen(s) && strncmp(s+i,"(",1) !=0;i++)
    ;
  if (strncmp(s+i,"(",1) ==0)
    p =insert(s,F);
  else
    p =insert(s,V);


  return p;
}


/*************************************************************
  push:
  called by decide.
  inserts a new binding into queue.
 *************************************************************/
void push(gen,arg,old,new,nt,ft)
int gen;
int arg;
int old;
int new;
int nt;
int ft;

{

  if (sp <MAXVAL )
    {
      queue[sp].gen = gen;
      queue[sp].arg = arg;
      queue[sp].old = old;
      queue[sp].new = new;
      queue[sp].nt = nt;
      queue[sp++].ft = ft;
    }
  else
    printf("error: stack full\n");
}

/*************************************************************
  pop:
  called by update2.
  retrieves a binding kept in queue for further update.
 *************************************************************/
struct rec pop()
{

  if (sp >0)
```

```
    {
      int i,j;
      i = queue[sp-1].gen;
      j = queue[sp-1].arg;
      return queue[--sp];
    }
  else
      printf("error: stack empty\n");

}
```