# 10 CLP Systems

In previous chapters we have introduced CLP languages and studied how to program with them. In this chapter we examine CLP languages from a different perspective: their implementation. This is not only of interest in its own right, but also allows the constraint programmer to gain a better understanding of the underlying efficiency issues.

We give a simple backtracking function for goal evaluation and then extend and refine its definition. One of the main implementation issues we will consider is the use of an incremental constraint solver to reduce the cost of constraint solving. We also show how to extend our evaluation mechanism to support the constructs for optimization, if-then-else, finding the first solution and negation introduced in Chapters 5, 7 and 9. The constructs for if-then-else, finding a first solution and negation are implemented in terms of a single primitive, the *cut*, while we give two possible implementations of the optimization construct.

## 10.1 Simple Backtracking Goal Evaluation

Previously we have understood goal evaluation to be a depth-first left-to-right search through the goal's derivation tree. The following algorithm performs such a search and provides a simplified view of the goal evaluation algorithm used in a CLP system. The function $defn_P(L)$ returns a sequence of rules defining the literal $L$ in order of their occurrence in the program $P$. Each rule is renamed away from all previously seen variables. The algorithm is a variant of the backtracking algorithm for solving CSPs (Algorithm 3.1). It returns the first answer it finds to the goal, but can easily be modified to find all of the answers to the goal. It is parametric in the choice of solver *solv* and simplifier *simpl*.

The algorithm first identifies the variables of interest, $W$, then calls the function simple_backtrack to find an answer, $C$, to the goal, and finally returns the answer simplified in terms of the variables of interest. Most of the work takes place in simple_backtrack which calls itself recursively to find the first answer to a state. If simple_backtrack is called with a state whose goal is empty, it returns the constraint store as the answer. If the goal is non-empty, simple_backtrack processes the first literal in the goal and then calls itself with the new state. If the literal is a primitive constraint it is added to the current constraint and the resulting constraint is

```
G and G1 are goals;
P is a program;
C and C1 are constraints;
L, L1, ... , Lk are literals;
and s1, ... , sn, t1, ... , tn are expressions.

simple_solve_goal(G)
    W := vars(G)
    C := simple_backtrack(⟨G | true⟩)
    return simpl(W, C)

simple_backtrack(⟨G | C⟩)
    if G is the empty goal then return C endif
    let G be of form L, G1
    cases
    L is a primitive constraint:
        if solv(C ∧ L) ≢ false then
            return simple_backtrack(⟨G1 | C ∧ L⟩)
        else return false endif
    L is a user-defined constraint:
        let L be of form p(s1, ... , sn)
        for each (p(t1, ... , tn) :- L1, ... , Lk) ∈ defnP(L) do
            C1 := simple_backtrack(⟨s1 = t1, ... , sn = tn, L1, ... , Lk, G1 | C⟩)
            if C1 ≢ false then return C1 endif
        endfor
        return false
    endcases
```

**Figure 10.1** Simple backtracking goal solver.

checked for satisfiability by the solver. If the literal is a user-defined constraint, it is rewritten using a rule from its definition. When evaluating a user-defined constraint, the rules are considered one by one, in the order returned by $defn_P$. This encodes a depth-first search.

**Algorithm 10.1:** Simple backtracking solver for goals
INPUT: A goal $G$ and a program $P$.
OUTPUT: An answer to $G$ for $P$.
METHOD: The algorithm is given in Figure 10.1. □

***Example 10.1***
As an example of how the simple backtracking solver works, consider the following CLP program to sum the numbers from 1 to $N$ introduced in Section 7.2:

```
sum(0, 0).
sum(N, N + S) :- sum(N-1, S).
```

Consider evaluation of the goal `sum(1,S)`. The initial call to simple_solve_goal sets the variables of interest, $W$, to $\{S\}$ and then calls simple_backtrack to evaluate

the state $\langle sum(1,S) \mid true \rangle$. The goal $sum(1,S)$ is treated as being of the form "$sum(1,S), \square$" where $\square$ is the empty goal. Thus $L$ is set to $sum(1,S)$ and $G_1$ to $\square$. As $L$ is a user-defined constraint, the second case statement is evaluated. The first rule defining $L$ is the fact `sum(0,0)`. This is shorthand for the rule

```
sum(0,0) :- □.
```

Thus, simple_backtrack is called recursively to evaluate the state

$$\langle 1 = 0, S = 0 \mid true \rangle.$$

In this call $L$ is set to $1 = 0$ and $G_1$ to $S = 0$. As $L$ is a primitive constraint, it is added to the current constraint store, giving $1 = 0$, and this is tested by the constraint solver for satisfiability. The solver returns $false$, so simple_backtrack returns $false$ from the call to evaluate $\langle 1 = 0, S = 0 \mid true \rangle$. The original call to simple_backtrack now tries the next rule defining $sum(1,S)$, effectively backtracking to the choicepoint associated with $sum(1,S)$ This rule, appropriately renamed, is

```
sum(N', N' + S')   :-   sum(N'-1, S').
```

Rewriting of $sum(1,S)$ with this rule leads to simple_backtrack being called to evaluate the state

$$\langle 1 = N', S = N' + S', sum(N' - 1, S') \mid true \rangle.$$

This processes the constraints $1 = N'$ and $S = N' + S'$ by adding them to the current constraint, and then calls simple_backtrack to evaluate

$$\langle sum(N' - 1, S') \mid 1 = N' \wedge S = N' + S' \rangle.$$

Evaluation of this call sets $L$ to $sum(N'-1,S')$ and $G_1$ to $\square$. The first rule defining $sum(N' - 1, S')$ is `sum(0,0)`. This leads to a call to simple_backtrack to evaluate

$$\langle N' - 1 = 0, S' = 0 \mid 1 = N' \wedge S = N' + S' \rangle.$$

Subsequent calls to simple_backtrack will process the constraints $N' - 1 = 0$ and $S' = 0$, adding them to the store. This leads to simple_backtrack being called to evaluate

$$\langle \square \mid 1 = N' \wedge S = N' + S' \wedge N' - 1 = 0 \wedge S' = 0 \rangle.$$

This simply returns the current constraint store, namely

$$1 = N' \wedge S = N' + S' \wedge N' - 1 = 0 \wedge S' = 0.$$

This constraint is returned from each of the nested calls to simple_backtrack, until finally it is returned from the original call. The function simple_solve_goal now returns $simpl(\{S\}, 1 = N' \wedge S = N' + S' \wedge N' - 1 = 0 \wedge S' = 0)$, which is simply $S = 1$, as we might hope.

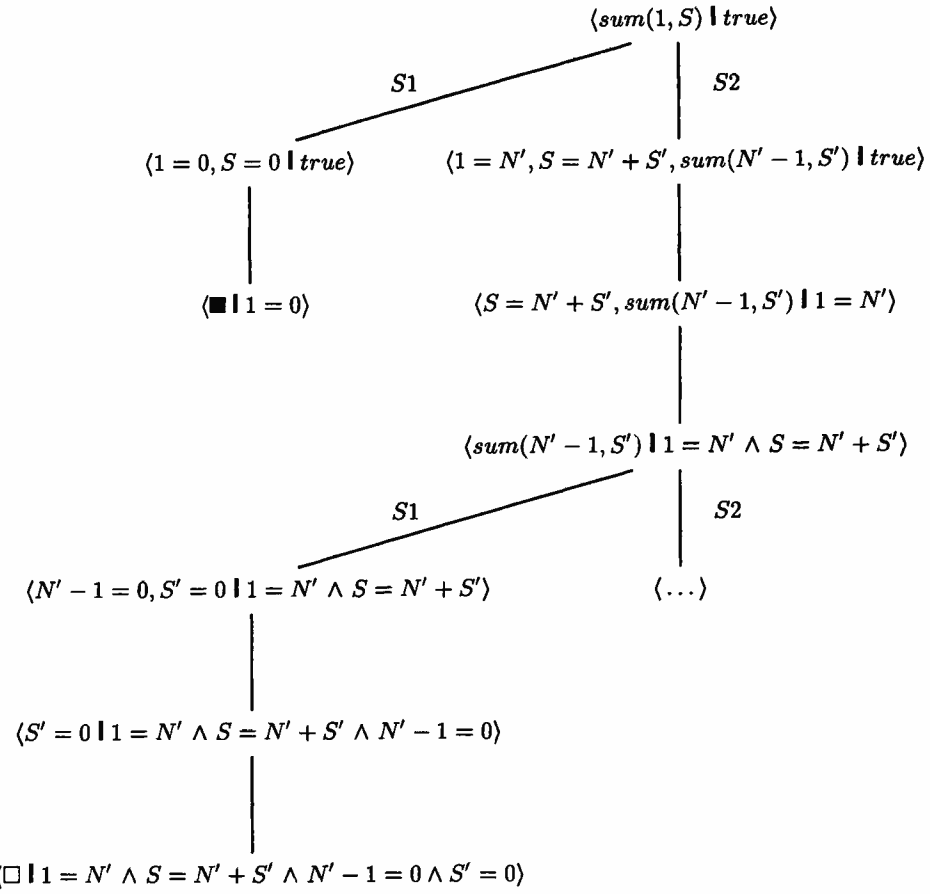



**Figure 10.2** Derivation tree for `sum(1,S)`.

We can understand the simple backtracking solver better if we examine Figure 10.2 which shows part of the derivation tree for `sum(1,S)`. We can see that the solver is performing a depth-first traversal of this tree, stopping when it reaches the first success state.

## 10.2 Incremental Constraint Solving

Unfortunately, the simple backtracking goal solver described above is inefficient. To see why, let us reconsider Example 10.1 and the evaluation of the goal `sum(1,S)`. When evaluating this goal, the constraint solver will be called on to determine satisfiability of each of the constraints in the following sequence:

$$1 = 0,$$
$$1 = N',$$
$$1 = N' \wedge S = N' + S',$$
$$1 = N' \wedge S = N' + S' \wedge N' - 1 = 0,$$
$$1 = N' \wedge S = N' + S' \wedge N' - 1 = 0 \wedge S' = 0.$$

Using the constraint solver to iteratively solve a sequence of constraints like this is not a very good idea, since each time the solver is called with one of the constraints it solves the problem from the beginning, even though it may be repeating work done when solving the previous constraint.

For this reason it is common in CLP systems to keep the current constraint in some sort of "partially solved form." This makes subsequent tests for satisfiability easier and primitive constraints are added to this as they are encountered during execution. Such a solver is said to be *incremental*, since it answers satisfiability questions incrementally by adding one more primitive constraint to an already satisfiable constraint.

***Definition 10.1***
An *incremental constraint solver*, *isolv*, for a constraint domain $\mathcal{D}$, takes as input a primitive constraint $c$ in $\mathcal{D}$ and returns *true*, *false* or *unknown*. There is an implicit global *constraint store*, $S$, which is manipulated by *isolv*. Whenever *isolv*($c$) returns *true*, constraint $S \wedge c$ must be satisfiable and whenever *isolv*($c$) returns *false*, the constraint $S \wedge c$ must be unsatisfiable. Additionally, whenever *isolv*($c$) returns *true* or *unknown*, the global constraint store must be updated to be equivalent to $S \wedge c$.

It is quite easy to modify many of the constraint solvers that we have met in Part 1 so that they become incremental. Consider the Gauss-Jordan constraint solver. As this already maintains a solved form $S$ and processes the primitive constraints one at a time, it is straightforward to give an incremental version. The algorithm is given below. The only trick is that we must remember to eliminate the non-parametric variables of $S$ from the new equality $c$, before processing $c$.

**Algorithm 10.2:** Incremental Gauss-Jordan solver
INPUT: A linear arithmetic equality $c$.
GLOBALS: $S$ is a conjunction of linear arithmetic equations in solved form.
OUTPUT: Returns *true* or *false* indicating whether $S \wedge c$ is satisfiable or not. Updates the global constraint store $S$.
METHOD: The algorithm is given in Figure 10.3. □

***Example 10.2***
Consider applying the incremental solver to the following equations which would be encountered when traversing the successful derivation in Example 10.1.

$$1 = N', \quad S = N' + S', \quad N' - 1 = 0, \quad S' = 0.$$

```
c is a linear arithmetic equation;
S is the global constraint store;
C is a conjunction of linear equations;
e, e1, ..., en are linear arithmetic expressions;
x, x1, ..., xn are real variables;
and r is a real number.

inc_gauss_jordan_solve(c)
    c := eliminate(c, S)
    if c can be written in form without variables then
        return eval(c)
    else
        write c in the form x = e where e does not involve x
        S := eliminate(S, x = e) ∧ x = e
        return true
    endif

eliminate(C, S)
    let S be of form x1 = e1 ∧ ··· ∧ xn = en
    for each xi do
        replace xi by ei throughout C
    endfor
    return C

eval(c)
    if c can be written in the form 0 = r where r ≠ 0 then
        return false
    else
        return true
    endif
```

**Figure 10.3** Incremental Gauss-Jordan solver.

Initially the global constraint store $S$ is *true*. First inc_gauss_jordan_solve is called with $1 = N'$. This is processed by first using *true* to eliminate variables from $1 = N'$. This simply gives $1 = N'$. As this contains variables, it is rewritten to $N' = 1$ and the constraint store becomes the result of using $N' = 1$ to eliminate variables from the old constraint store, *true*, and conjoining $N' = 1$. This is simply $N' = 1$. The call to the solver returns *true*, indicating that the store remains satisfiable when $1 = N'$ is added to it.

The second call is with the constraint $S = N' + S'$. The first step is to eliminate variables from $S = N' + S'$ using the current store $N' = 1$. This gives $S = 1 + S'$. As $S = 1 + S'$ contains variables, it is rewritten to $S = 1 + S'$ and the constraint store is updated by using $S = 1 + S'$ to eliminate variables from it and conjoining it with $S = 1 + S'$, giving $N' = 1 \wedge S = 1 + S'$. Note that we could also have chosen to rewrite the constraint to $S' = S - 1$. Finally, *true* is returned, indicating that the constraint store remains satisfiable when $S = N' + S'$ is added to it.

The next call is with $N' - 1 = 0$. First the current store is used to eliminate variables from it, giving $1 - 1 = 0$. As $1 - 1 = 0$ does not contain variables, it is evaluated to see if it is equivalent to *true*. As it is, *true* is returned and the constraint store remains unchanged. This is the expected behaviour since $N' - 1 = 0$ was redundant with respect to the store.

The last call to the incremental solver is with $S' = 0$. Eliminating variables from $S' = 0$ using the current store leaves the constraint unchanged. Now $S' = 0$ is used to eliminate $S'$ from the store and conjoined, setting the constraint store to $N' = 1 \wedge S = 1 + 0 \wedge S' = 0$. The value *true* is returned from the call to the solver indicating the store remains satisfiable.

Therefore the incremental Gauss-Jordan algorithm has efficiently determined that the following constraints

$$
\begin{aligned}
&1 = N', \\
&1 = N' \wedge S = N' + S', \\
&1 = N' \wedge S = N' + S' \wedge N' - 1 = 0, \\
&1 = N' \wedge S = N' + S' \wedge N' - 1 = 0 \wedge S' = 0
\end{aligned}
$$

are satisfiable. To appreciate the importance of incremental constraint solving, the reader is encouraged to solve each of these constraints in turn with the non-incremental Gauss-Jordan algorithm and compare the amount of work involved.

The following algorithm captures the goal evaluation algorithm used in a CLP system which uses an incremental solver *isolv* that works on a global constraint store. The evaluation function inc_backtrack is similar to simple_backtrack. However, since the constraint component of the state being evaluated is now implicitly available in the global constraint store, it only takes a goal as an argument. Also, since the answer is in the global constraint store, it simply returns *true* or *false* to indicate whether an answer has been found or not. The main difference is that inc_backtrack must now explicitly restore the global constraint store to its previous value when a new rule is tried in the rewriting of a user-defined constraint. The function initialize_store is used to initialize the store so that it represents *true*, the function get_store returns the constraint represented by the global constraint store and the functions save_store and restore_store, respectively, save the current configuration of the store and then restore it.

**Algorithm 10.3:** Incremental backtracking solver for goals
INPUT: A goal $G$.
GLOBALS: A constraint store $S$ and a program $P$.
OUTPUT: An answer to $G$ for $P$ or *false* if none exist.
METHOD: The algorithm is given in Figure 10.4. □

The easiest way to implement save_store and restore_store is by means of a global stack called the *backtrack stack*. The function save_store simply pushes the current configuration of the store on to the stack, while restore_store pops the old

```
G and G_1 are goals;
P is a program;
L, L_1, ..., L_k are literals;
s_1, ..., s_n, t_1, ..., t_n are expressions;
and W is a set of variables.

inc_solve_goal(G)
    W := vars(G)
    initialize_store()
    if inc_backtrack(G) then
        return simpl(W, get_store())
    else
        return false
    endif

inc_backtrack(G)
    if G is the empty goal then return true endif
    let G be of form L, G_1
    cases
    L is a primitive constraint:
        if isolv(L) ≠ false then
            return inc_backtrack(G_1)
        else return false endif
    L is a user-defined constraint:
        let L be of form p(s_1, ..., s_n)
        for each (p(t_1, ..., t_n) :- L_1, ..., L_k) ∈ defn_P(L) do
            save_store()
            if inc_backtrack((s_1 = t_1, ..., s_n = t_n, L_1, ..., L_k, G_1)) then
                return true
            endif
            restore_store()
        endfor
        return false
    endcases
```

**Figure 10.4** Incremental backtracking goal solver.

configuration from the top of the stack. We shall explore better implementation techniques in the next section.

***Example 10.3***

As an example of the way in which the incremental backtracking goal solver works, consider its operation on the goal and program from Example 10.1. We shall use the incremental Gauss-Jordan algorithm for constraint solving.

The call to evaluate the goal `sum(1,S)` initializes the global constraint store to *true*, sets the variables of interest, $W$, to $\{S\}$ and then calls inc_backtrack to evaluate $sum(1, S)$ in the context of the global constraint store. As before, $L$ is set to $sum(1, S)$ and the second case is used. The first rule defining $sum(1, S)$ is `sum(0,0)`. The current store is saved on to the initially empty backtrack stack

giving the stack:

| *true* |
|---|

Now the goal "$1 = 0, S = 0$" is evaluated using inc_backtrack. $L$ is set to $1 = 0$ and $G_1$ to $S = 0$. As $L$ is a primitive constraint, the incremental solver is called to add $1 = 0$ to the constraint store. This returns *false*, so the call to evaluate "$1 = 0, S = 0$" returns *false*. This means that inc_backtrack must backtrack to the last choicepoint. This is done by calling restore_store, which pops the constraint *true* from the backtrack stack and sets the global constraint store to *true*.

Evaluation of $sum(1, S)$ now tries the next rule defining $sum(1, S)$. This is

```
sum(N', N' + S')   :-  sum(N' - 1, S').
```

First the current store is saved, giving the backtrack stack:

| *true* |
|---|

Next the goal "$1 = N', S = N' + S', sum(N' - 1, S')$" is evaluated using inc_backtrack. This uses the incremental solver to process the constraint $1 = N'$ and then calls itself recursively to evaluate the goal "$S = N' + S', sum(N' - 1, S')$." Similarly, this call uses the incremental solver to process $S = N' + S'$ and then calls itself recursively to evaluate $sum(N' - 1, S')$. At this stage, the current store is $N' = 1 \wedge S = 1 + S'$ and the backtrack stack has not been affected. The first rule defining $sum(N' - 1, S')$ is `sum(0,0)`. The constraint store is now saved, giving the stack

| $N' = 1 \wedge S = 1 + S'$ |
|---|
| *true* |

and the goal "$N' - 1 = 0, S' = 0$" is now evaluated.

The procedure inc_backtrack processes the constraints $N' - 1 = 0$ and $S' = 0$ in turn, as detailed in Example 10.2, setting the global constraint store to

$$N' = 1 \wedge S = 1 + 0 \wedge S' = 0.$$

The backtrack stack is not affected. As all literals have been processed, the call finishes and simply returns *true*. In turn, *true* is returned from each of the nested calls, until the original call to evaluate `sum(1,S)` returns. Finally, inc_solve_goal simplifies the constraint store in terms of the variables of interest and returns the result $S = 1$.

## 10.3 Efficient Saving and Restoring of the Constraint Store

One consequence of using an incremental solver is the need to explicitly save the constraint store when a choicepoint is reached and then to restore the store when backtracking to that choicepoint. Doing this efficiently is an important issue in the implementation of CLP languages and has implications for the design of the incremental solver.

Naively saving and restoring the entire constraint store by pushing and popping it to and from a stack is expensive in both space and time. Fortunately, it is not really necessary to do this. Instead, we need only save enough information to allow recomputation of the old constraint store from the current constraint store. Imagine that we are at a choicepoint and wish to save the current store $C$. Now we add the primitive constraints $c_1, \ldots, c_n$ to $C$, giving an unsatisfiable store $C'$. We must backtrack to the choicepoint and restore $C$. It is not necessary to have stored $C$, rather we need only to have kept sufficient information, $\delta$, such that we can compute $C$ from $C'$ and $\delta$. That is, $\delta$ needs to contain enough information to "undo" the effect of adding $c_1, \ldots, c_n$ to $C$. Exactly what information needs to be kept in $\delta$ depends on the particular constraint solver.

We now briefly discuss two choices of $\delta$ and the corresponding methods for restoring the store that are used in constraint logic programming systems.

### 10.3.1 Trailing

One simple technique for restoring the store is to associate an index with each constraint as it is added, in effect *time stamping* the constraint. At a choicepoint we can store the current time stamp and, on backtracking, remove all constraints with a larger time stamp since they must have been added after the choicepoint. However, this does not take into account the fact that adding new constraints may change the form of the old constraints, as, for instance, when the new constraint causes a variable to be eliminated. To handle this we can, whenever an old constraint is modified, save the old value of that part of the constraint which has been modified and restore this on backtracking. This is called *trailing*. Note that, if part of a constraint is repeatedly modified, we need only save the old value once for each choicepoint.

To clarify this discussion, consider the incremental Gauss-Jordan solver. To allow constraints to be easily time stamped, primitive constraints are kept in an indexed sequence. Whenever a constraint is added to the solver it is added to the end of the sequence. When a choicepoint is set up, the index of the last constraint in the store, *last*, is placed on the backtrack stack and the trail associated with that element is initialised. Whenever a primitive constraint is added to the store it may cause earlier constraints to be modified. Constraints may be modified either by changing the coefficient, $a$, of a variable $x$ to $a'$, say, or by changing the constant, $b$, to $b'$, say. Whenever constraint $i$ is modified, if $i \leq last$, an entry of the form $\langle i, x, a \rangle$

or $\langle i, constant, b \rangle$ is added to the trail associated with the last choicepoint on the backtrack stack. However, if there is already an entry on the trail for that position, there is no need to add another entry as the value to be restored is already on the trail.

Recall the sum program from Example 10.3 and consider evaluation of the goal

```
sum(1,S), S=2.
```

Initially, both the backtrack stack and store are empty. The first choicepoint occurs when deciding with which rule to rewrite $sum(1, S)$. The first rule tried is `sum(0,0)`. At this stage the current constraint store is empty, that is *true*, so the index of the last constraint in the store is 0. The call to save_store will push this on to the backtrack stack, together with an initially empty trail ([]), giving

| 0 | [] |
|---|---|

Next we process $1 = 0$. This returns *false*, causing evaluation to backtrack. The call to restore_store pops 0 and the associated empty trail from the backtrack stack. It then removes all constraints with index greater than 0 from the constraint store, leaving the empty store, *true*. As the trail is empty, the store has been restored.

We now try the next rule defining `sum(1,S)`. This rule is

```
sum(N', N' + S')   :-   sum(N' - 1, S').
```

Again the current store is saved, giving the backtrack stack:

| 0 | [] |
|---|---|

Now the constraints $1 = N'$ and $S = N' + S'$ are processed by adding them to the constraint store. Next $sum(N' - 1, S')$ is evaluated. The first rule defining $sum(N' - 1, S')$ is `sum(0,0)`. We must save the constraint store. This pushes the index of the last constraint in the store, 2, on to the backtrack stack. The backtrack stack and constraint store are now:

| 2 | [] |
|---|---|
| 0 | [] |

1: $N' = 1 \wedge$
2: $S = 1 + S'$

Note that we have explicitly written the index of the constraints in the store on the left.

Now the constraints $N' - 1 = 0$ and $S' = 0$ are processed. Processing $N' - 1 = 0$ does not change the store. Adding $S' = 0$ changes the store to

$$N' = 1 \wedge S = 1 \wedge S' = 0.$$

The constraint $S = 1 + S'$ has been modified to $S = 1$. Since the index of this constraint is 2, which is less than or equal to the index of the top entry on the backtrack stack, the associated trail must be modified to remember the old

coefficient of $S'$. The backtrack stack and constraint store become

| | |
|---|---|
| 2 | $[\langle 2, S', 1\rangle]$ |
| 0 | [] |

$$\begin{aligned} 1:&\quad N' = 1 \wedge \\ 2:&\quad S = 1 \wedge \\ 3:&\quad S' = 0. \end{aligned}$$

At this point, the call $sum(1, S)$ has been evaluated. Now the constraint $S = 2$ is added to the store. This causes evaluation to backtrack. The call to restore_store first pops the index 2 and the associated trail $\langle 2, S', 1\rangle$ from the backtrack stack. Then all constraints with index greater than 2 are removed from the constraint store, leaving $N' = 1 \wedge S = 1$. Finally, the entries on the trail are traversed. In this case, the single entry is used to change the coefficient of $S'$ from 0 back to 1 in the constraint with index 2. Thus, the backtrack stack and constraint store become:

| | |
|---|---|
| 0 | [] |

$$\begin{aligned} 1:&\quad N' = 1 \wedge \\ 2:&\quad S = 1 + S' \end{aligned}$$

Subsequent evaluation will rewrite $sum(N' - 1, S')$ with the second rule and lead to an infinite derivation.

### 10.3.2 Semantic Backtracking

The combination of time stamping and trailing is a general technique for saving and restoring the constraint store. However, for particular constraint domains it may be possible to use approaches which do not explicitly store the changes made to the constraint store but, rather, store the high-level operations which have been performed on the store and undo these operations. Such approaches to backtracking are said to be *semantic*.

As an example of semantic backtracking, consider the incremental Gauss-Jordan algorithm yet again. When a new constraint is added, the only way the old constraints in the store can be affected is if the new constraint is used to eliminate a variable, $x$, say. We can undo the effect of this elimination if we remember the old coefficient of $x$ in each of these constraints.

For instance, imagine that the constraint store, $C$, is:

$$\begin{aligned} 1:&\quad X = Y + 2Z + 4 \wedge \\ 2:&\quad U = 3Y + Z - 1 \wedge \\ 3:&\quad V = 3. \end{aligned}$$

And suppose that we add the constraint $Y + 2V + X = 2$.

After we eliminate variables from the new constraint we obtain $2Y + 2Z = -8$. Now imagine that we use this constraint to eliminate $Y$. This results in the

constraint store, $C'$,

$$\begin{aligned} 1: &\quad X = Z \wedge \\ 2: &\quad U = -2Z - 13 \wedge \\ 3: &\quad V = 3 \wedge \\ 4: &\quad Y = -Z - 4. \end{aligned}$$

To make it possible to undo the effect of eliminating $Y$ from the old constraint $C$, we must remember the old coefficient of $Y$ in each primitive constraint in $C$. This can be stored in a list of tuples consisting of the constraint index and the coefficient. For efficiency, constraints with a 0 coefficient are not placed in the list. Thus we must remember the list

$$[\langle 1, 1\rangle, \langle 2, 3\rangle].$$

Now imagine that we wish to undo the effect of adding $Y + 2V + X = 2$ to the store. We first undo the effect of using $Y = -Z - 4$ to eliminate $Y$ from each of the old constraints. This is achieved by multiplying $Y + Z + 4$ by the original coefficient of $Y$ and adding it to the right hand side of the equation. That is, the first equation becomes $X = Z + 1 \times (Y + Z + 4)$ which is $X = Y + 2Z + 4$, the second equation becomes $U = -2Z - 13 + 3 \times (Y + Z + 4)$ which is $U = 3Y + Z - 1$ and the third equation remains unchanged since the coefficient is 0. Finally, we remove the last constraint, $Y = -Z - 4$, from the store giving $C$ as before:

$$\begin{aligned} 1: &\quad X = Y + 2Z + 4 \wedge \\ 2: &\quad U = 3Y + Z - 1 \wedge \\ 3: &\quad V = 3. \end{aligned}$$

Thus, using only the coefficient list, we have been able to recompute $C$ from $C'$.

From this example we can see that we can undo the effect of adding a primitive constraint $c$ to a store $C$ by remembering the coefficients of the variable $c$ is used to eliminate. If $c$ is redundant or leads to *false* we do not need to remember a coefficient list, as the old constraint remains unchanged.

The advantage of this particular semantic backtracking technique over trailing is that the size of the coefficient list is usually quite small and often requires less space than using a trail. The disadvantage is that restoring the store requires more computation, essentially performing an elimination in reverse, so restoration of the store using a trail is faster.

## 10.4 Implementing If-Then-Else, Once and Negation

So far we have restricted ourselves to "pure" constraint logic programs as introduced in Chapter 4. However in Chapters 7 and 9 we introduced three additional programming constructs for if-then-else, finding the first solution and negation. We now investigate how these are implemented in a CLP system.

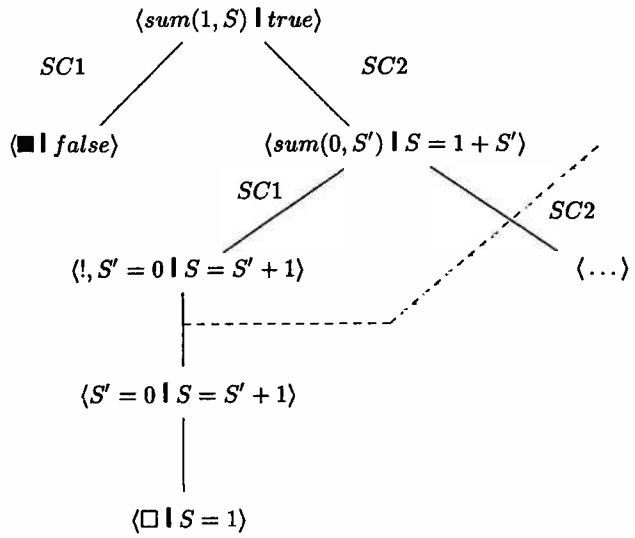



**Figure 10.5** Simplified derivation tree for `sum(1,S)`.

In fact all three are implemented by using a single new language construct the *cut*, written `!`. The cut is used to prune derivations from the derivation tree and works by directing the system to remove previously set-up choicepoints. Cuts are very powerful, but should be used sparingly. We did not introduce them directly since in almost all cases the cut can be replaced by appropriate if-then-else, once or negative literals. In these cases it is preferable to use the appropriate high-level construct rather than the cut.

***Example 10.4***

Consider the improved sum program from Chapter 7.

```
sum(0,0).                                          (S7)
sum(N, N + S)   :-  N ≥ 1, sum(N - 1,S).           (S8)
```

For the mode of usage in which the first argument is fixed, we can rewrite the program using an if-then-else literal to:

```
sum(N, SS) :-
     (N = 0 ->
          SS = 0
     ;
          N ≥ 1, SS = N + S, sum(N - 1,S) ).
```

This program is equivalent to the following program which uses the cut.

```
sum(N, SS).   :-  N = 0, !, SS = 0                 (SC1)
sum(N, SS)    :-  N ≥ 1, SS = N + S, sum(N - 1,S).  (SC2)
```

```
L is a user-defined constraint:
    let L be of form p(s_1, ..., s_n)
    for each (p(t_1, ..., t_n) :- L_1, ..., L_k) ∈ defn_P(L) do
        i := save_store()
        if for some 1 ≤ j ≤ n, L_j ≡ ! then
            if inc_backtrack(s_1 = t_1, ..., s_n = t_n, L_1, ..., L_{j-1}) then
                remove_choicepoints(i)
                return inc_backtrack(L_{j+1}, ..., L_k, G_1)
            endif
        elseif inc_backtrack(s_1 = t_1, ..., s_n = t_n, L_1, ..., L_k, G_1) ≡ true then
            return true
        endif
        restore_store()
    endfor
    return false
```

**Figure 10.6** Modification of incremental goal solver for cut.

The cut indicates that, once it has been reached, there is no need to consider the other rule in the definition of sum. For the mode of usage we are interested in this is correct since the rules are *mutually exclusive*. That is, a fixed value of $N$ cannot simultaneously satisfy the constraint $N = 0$ and the constraint $N \geq 1$. Thus, once we know that a particular value of $N$ does satisfy $N = 0$, we know the second rule will fail with this value of $N$ and, so, we can remove the choicepoint set up for sum.

Examining the derivation tree for the goal `sum(1, S)` shown in Figure 10.5 we can see the effect of the cut. When the cut literal is selected, every branch remaining to the right, up to the state just before the cut was introduced is pruned. These branches will never be explored in the evaluation. For this example, the second rule $SC2$ is never used to rewrite the state $\langle sum(0, S') \mid S = S' + 1 \rangle$ because of the cut.

A cut commits the system to all choices which have been made since the parent goal was rewritten with the rule in which the cut appears. To understand the precise definition of the cut, consider the following program fragment.

A :- $L_1$, $L_2$, ... , $L_i$, !, $L_{i+2}$, ... , $L_n$.

Imagine that this rule has been used to rewrite the literal $A_{call}$ in the current goal and that the literals $L_1$ to $L_i$ have subsequently been successfully evaluated, perhaps with some backtracking local to their evaluation. When the cut is evaluated, the system commits to rewriting $A_{call}$ with the rule in which the cut appears. That is to say, the choicepoint associated with the rewriting of $A_{call}$ is removed and the remaining rules in the definition of $A_{call}$ will not be considered. Furthermore, all choicepoints placed when evaluating $L_1$ to $L_i$ are removed. If, on subsequent evaluation the derivation fails, the system will backtrack to choicepoints constructed before $A_{call}$ was rewritten.

It is quite simple to extend the incremental backtracking goal solver of Figure 10.4 to handle cuts. The modification to the case for evaluating a user-defined constraint is shown in Figure 10.6. For simplicity, we assume there is, at most, one cut in a rule.

Since we need to remove choicepoints when a cut is encountered, we must modify save_store so that it returns an index and provide a function remove_choicepoints($i$) which removes all stores on the backtrack stack that have index $i$ or greater.

***Example 10.5***

Consider the execution of the goal h(X) using the program:

```
h(X)  :-  X ≥ 0, p(X), q(X).   (H1)
h(4).                          (H2)
p(X)  :-  X ≤ 4, r(X), !.      (P1)
p(3).                          (P2)
r(1).                          (R1)
r(2).                          (R2)
q(2).                          (Q1)
q(3).                          (Q2)
```

The simplified derivation tree is shown in Figure 10.7. Execution proceeds by calling inc_backtrack to evaluate h(X). This saves the store *true* with index 1. After collecting the constraint $X \geq 0$, inc_backtrack is called to evaluate "$p(X), q(X)$." The store $X \geq 0$ is saved with index 2. Since the rule $P1$ includes a cut, the literals before the cut "$X \leq 4$, $r(X)$" are evaluated first. The constraint $X \leq 4$ is collected and subsequent evaluation of r(X) saves the store $X \geq 0 \wedge X \leq 4$ on the stack with index 3. The rule $R1$ succeeds, so the store becomes $X \geq 0 \wedge X \leq 4 \wedge X = 1$, which is simplified to $X = 1$ in the figure. Thus, the call to inc_backtrack to evaluate $r(X)$ returns *true* and so in turn, the call to evaluate "$X \leq 4, r(X)$" returns *true*. Since the cut has been seen, remove_choicepoints(2) is called. This pops those stores with index greater than or equal to 2 off the backtrack stack. Thus the stack will be left with only the store true with index 1. Now the remainder of the goal, the literal q(X), is evaluated. This saves the store $X = 1$ with index 2, and then fails using rule Q1. Backtracking restores the store to $X = 1$ and tries rule $Q2$ which also fails. Here, the call to inc_backtrack to evaluate $q(X)$ returns *false*, and so the call to evaluate "$X \geq 0, p(X), q(X)$" returns *false*. The store is restored to *true* (since once we are within the call to inc_backtrack to evaluate $h(X)$) and the second rule for $h$ is tried which gives the answer $X = 4$. Note that, because of the cut, the rules $P2$ and $Q2$ were not tried.

Using meta-level programming, it is straightforward to define negation, if-then-else and once in terms of cut. The simplest is the definition of once. It is

```
once(G) :- call(G), !.
```

The effect is as follows. The goal G is evaluated and if an answer is found, this answer is committed to.

The definition of not is also reasonably simple:

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

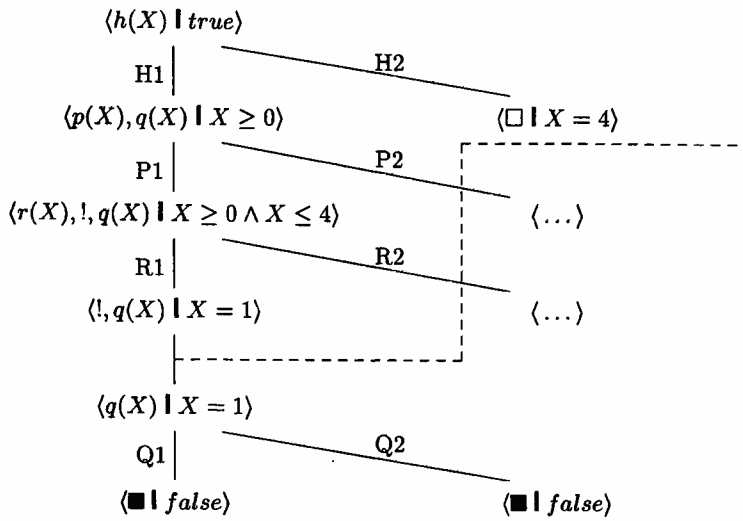



**Figure 10.7** Simplified derivation tree for `h(X)`.

The effect is as follows. The goal $G$ will be executed and, if an answer is found, the first rule will be committed to and then it will fail. If $G$ finitely fails, the second rule, which always succeeds, will be used.

***Example 10.6***

Consider evaluation of the goal `not(a=b)`. Evaluation of the negative literal uses the first rule, calling the goal `a=b`. This fails, so the second rule for `not` is tried, which will succeed.

Now consider evaluation of the goal `not(a=a)`. Evaluation of the negative literal initially tries the first rule, calling the goal `a=a`. This succeeds, so evaluation of the cut removes the choicepoint corresponding to the call `not(a=a)`, in effect committing to the first rule. Now the literal `fail` is evaluated. Thus the first rule fails, and, since the cut has been evaluated, the second rule is not tried. Thus, evaluation of `not(a=a)` fails.

The definition of if-then-else is slightly more complex, but still reasonably intuitive.

```
(G1 -> G2 ; G3) :- call(G1), !, call(G2).
(G1 -> G2 ; G3) :- call(G3).
```

The effect is, if $G1$ has a solution, then the first rule and this particular solution to $G1$ are committed to and $G2$ is executed. Otherwise, if $G1$ finitely fails, the second rule is tried and $G3$ is executed.

The reader is encouraged to verify that these definitions, together with the incremental backtracking goal solver augmented to handle cuts, produce the same behaviour as the conceptual definition of the if-then-else, once and negation constructs given earlier.

## 10.5 Optimization

Another important language construct provided by most CLP languages is optimization. The idealized optimization construct we have used in this text is `minimize(G, E)`. Like other optimization constructs, `minimize` can be implemented in a number of different ways, no one of which is uniformly more efficient. For this reason many CLP systems provide more than one minimization construct, with different constructs corresponding to different implementation techniques. Here we will discuss two of the simplest methods for implementing `minimize`.

Consider the evaluation of `minimize(G, E)` in the context of constraint store $C$. Implementation of `minimize` can be split into three parts.

The first part is a domain specific optimization procedure, minimize_store($E$). This minimizes the value of the expression $E$ with respect to the current constraint store. For real linear constraints a natural choice is to use the simplex algorithm for optimization and to give a run-time error if the expression to be minimized is not linear. For finite constraint domains the expression is usually required to be fixed by the store, so no "real" optimization is necessary.

The second part of the implementation is an algorithm to search through the derivation tree of the state $\langle G \mid C \rangle$ in order to find the minimum value, $m$, of the expression $E$. This part makes use of minimize_store($E$) whenever an "interesting" answer to $\langle G \mid C \rangle$ is found.

The final part of the implementation generates the answers to `minimize(G, E)` given that the minimum value, $m$, of $E$ is known. This may be done by simply returning the answers to the state $\langle E = m, G \mid C \rangle$.

Different approaches to minimization primarily differ in the way in which they search through the derivation tree to find the minimum value. Here we will give two of the simplest approaches. Each corresponds to one of the approaches to finding the optimal solution to an arithmetic CSP integer which were discussed in Section 3.6.

The simplest approach is analogous to retry_int_opt. The idea is to search for an answer, $C_1$, to the state $\langle G \mid C \rangle$ and then use the domain specific optimization procedure to find the minimal value $m$ of $E$ with respect to $C_1$. Now we search for an answer to the state $\langle G \mid C \wedge E < m \rangle$. If an answer is found, we update the value of $m$ and repeat the process. If there is no answer, then the minimum value has been found.

It is straightforward to modify the incremental goal solver of Figure 10.4 so as to handle minimization literals using this "retrying" technique. We simply add another case to the case statement for minimization literals. The code for this is shown in Figure 10.8.

Recall the program for the butterfly combination of call options given in Section 5.2. Let us examine how the goal

```
minimize( butterfly(S,P), -P).
```

is evaluated. The simplified derivation tree for the goal `butterfly(S, P)` is shown

```
L is a minimization literal:
    let L be of form minimize(G, E)
    i := save_store()
    m := +∞
    while inc_backtrack(E < m, G) ≢ false do
        m := minimize_store(E)
        remove_choicepoints(i + 1)
        restore_store()
        i := save_store()
    endwhile
    restore_store()
    return inc_backtrack((E = m, G, G1))
```

**Figure 10.8** Modification of incremental goal solver for retry optimization.

in Figure 10.9. For brevity the goals $G_1, G_2$ and $G_3$ are defined to be

$$G_1 = \mathit{call\_option}(1, 100, 500, S, P1), \mathit{call\_option}(-1, 200, 300, S, P2),$$
$$\mathit{call\_option}(1, 400, 100, S, P3)$$
$$G_2 = \mathit{call\_option}(-1, 200, 300, S, P2), \mathit{call\_option}(1, 400, 100, S, P3)$$
$$G_3 = \mathit{call\_option}(1, 400, 100, S, P3).$$

Execution of the goal `minimize(butterfly(S,P), -P)` begins by calling inc_backtrack$(-P < +\infty, \mathit{butterfly}(S, P))$. This discovers the first answer,

$$-P < +\infty \wedge P = -100 \wedge 0 \leq S \leq 1.$$

Minimizing $-P$ with respect to this store gives $m = 100$. The store is restored to the state it was before evaluating "$-P < +\infty, \mathit{butterfly}(S, P)$" and inc_backtrack is called to evaluate "$-P < 100, \mathit{butterfly}(S, P)$." The first answer found (corresponding to the second answer in the derivation tree for `butterfly(S, P)`) is

$$-P < 100 \wedge P = 100S - 200 \wedge 1 \leq S \leq 3.$$

Calling minimize_store to minimize $-P$ with respect to this store gives $m = -100$. Again the store is restored and now "$-P < -100, \mathit{butterfly}(S, P)$" is evaluated. This fails. Hence, $m = -100$ is the minimum value for $-P$. Execution proceeds by calling inc_backtrack$(-P = -100, \mathit{butterfly}(S, P))$. This returns the first answer $P = 100 \wedge S = 3$ (corresponding to the derivation in Figure 10.9 with answer $P = 100S - 200 \wedge 1 \leq S \leq 3$). A second identical answer (corresponding to the derivation in Figure 10.9 with answer $P = -100S + 400 \wedge 3 \leq S \leq 5$) also exists.

Notice that, in calculating the minimal value for $-P$, we repeatedly (in effect), searched the derivation tree for `butterfly(S,P)`. Overall the execution visited states corresponding to those shown in the tree in Figure 10.9 approximately 25 times.

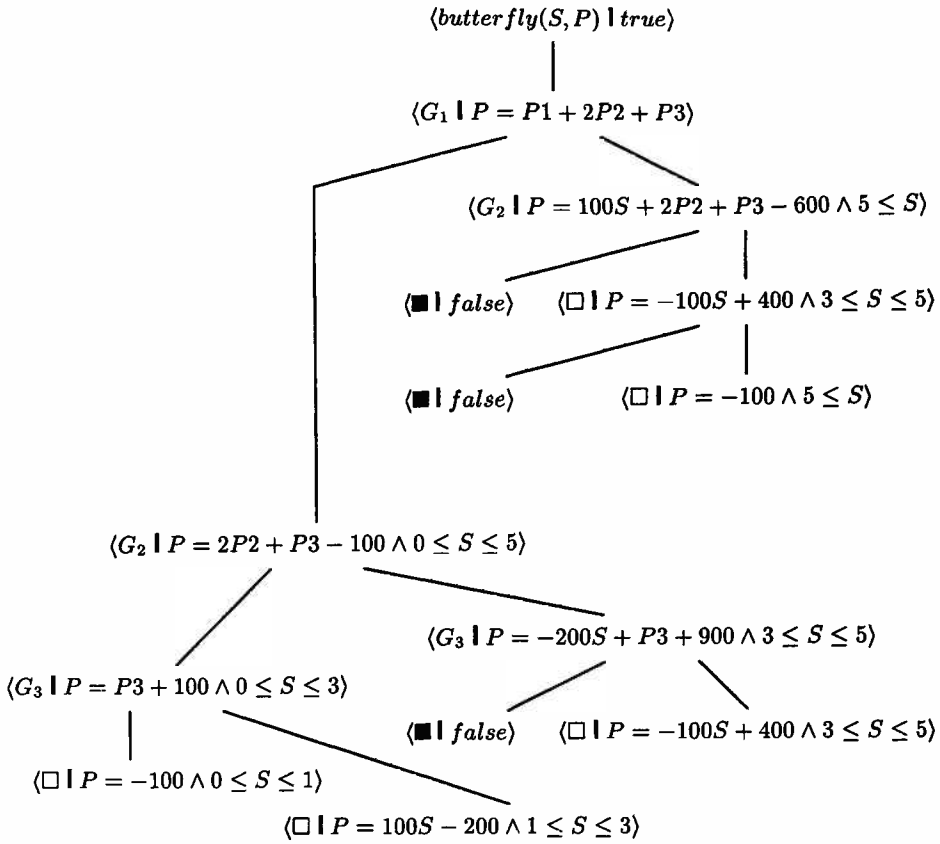



**Figure 10.9** Derivation tree for `butterfly(S, P)`.

The second approach to searching through the derivation tree for $G$ in order to find the minimal value of $E$ is analogous to the approach used in backtrack_int_opt. Only a single backtracking search of the derivation tree for $G$ is made, and the minimal value for $E$ found so far is stored in $m$. Whenever we find a new solution we test whether it is better than the previous one and, if so, update $m$.

The modification to the incremental goal solver of Figure 10.4 to handle minimization by backtracking is shown in Figure 10.10. It adds two cases to the case statement, one to handle minimization subgoals and one to handle *catch* subgoals. The case for minimization subgoals initializes $m$ and sets up a dummy subgoal $catch(m, E)$ which is reached after a solution to $G$ is found. The case for *catch* subgoals always fails, but updates $m$ to hold the minimum value found so far.[1]

1. Unfortunately this program will not work for nested minimization subgoals since $m$ is being used as a global variable. To fix this problem we could use a stack of $m$s.

```
L is a minimization subgoal:
    let L be of the form minimize(G, E)
    i := save_store()
    m := +∞
    inc_backtrack(G, catch(m, E))
    restore_store(i)
    return inc_backtrack((E = m, G, G'))
L is a catch subgoal:
    let L be of the form catch(m, E)
    if isolv(E < m) ≠ false then
        m := minimize_store(E)
    endif
    return false
```

**Figure 10.10** Modification of incremental goal solver for backtracking optimization.

Execution of the goal `minimize( butterfly(S,P), -P)` begins by storing the solver state, initializing $m$ to $+\infty$ and calling inc_backtrack to evaluate "$butterfly(S,P), catch(m,-P)$". Evaluation traverses the leftmost branch finding the answer $P = -100 \wedge 0 \leq S \leq 1$ to `butterfly(S,P)`. Now the *catch* subgoal is evaluated. This adds the constraint $-P < +\infty$ which succeeds and so $m$ is set to 100. Execution continues by backtracking to find the next answer. This is $P = 100S - 200 \wedge 1 \leq S \leq 3$. Again the *catch* subgoal is reached. Adding the constraint $-P < 100$ succeeds so $m$ is set to $-100$. The next time the *catch* subgoal is reached the store is $P = -100S + 400 \wedge 3 \leq S \leq 5$. Adding the constraint $-P < -100$ fails, so $m$ is not updated. Similar behaviour occurs with the next answer found, $P = -100 \wedge S \geq 5$. After the entire derivation tree for `butterfly(S,P)` has been explored, $m$ has been calculated to be $-100$. Execution proceeds as before using inc_backtrack$(-P = -100, butterfly(S,P))$. In this case only 13 visits are made to states corresponding to those shown in Figure 10.9, as opposed to retry optimization which made 25 visits.

An even better method for finding the minimum value of the expression $E$, is to modify the above approach so that it adds the constraint $E < m$ to the constraint store whenever a call to restore_store occurs in the execution of the call inc_backtrack$(G, catch(m, E))$. For simplicity, we omit this definition.

## 10.6 Other Incremental Constraint Solvers

In Section 10.2 we saw the importance of incremental constraint solving in the case of linear arithmetic equations. The same arguments hold for any constraint solver used within a CLP system. In this section we show how the tree constraint solver, tree_solve, of Section 1.4 and the bounds consistency solver, bounds_solve, of Section 3.4 can be made incremental.

### 10.6.1 Incremental Tree Constraint Solving

We first look at how to define an incremental solver for tree constraints. As for the incremental Gauss-Jordan solver, the key is to use a solved form representation of the primitive constraints encountered so far. Recall that execution of the tree constraint solver algorithm, unify, from Section 1.4 returns either *false* or a tree constraint in solved form

$$x_1 = t_1 \wedge \ldots \wedge x_n = t_n,$$

where each $x_i$ is a distinct variable not occurring in any of the $t_i$.

Given the solved form $x_1 = t_1 \wedge \ldots \wedge x_n = t_n$, we can use the function eliminate defined in Figure 10.3 to simplify another tree constraint $C$ by eliminating the variables $x_1, \ldots, x_n$ from $C$. Although eliminate was originally defined for eliminating real variables using linear equations, the same process can be used for term equations in solved form. The function eliminate replaces every occurrence of a variable $x_i$ in $C$ by the term $t_i$. For instance, if $S$ is the solved form $X = f(V) \wedge Y = V$ and $C$ is $f(W) = f(X) \wedge X = f(a)$, then eliminate$(C, S)$ is

$$f(W) = f(f(V)) \wedge f(V) = f(a).$$

Elimination allows us to compute the solved form incrementally. Imagine that we first encounter the equation $c_0$ which is $g(X, Y) = g(f(V), V)$. We compute the solved form, $S_0$, of $c_0$ which is $X = f(V) \wedge Y = V$. Suppose we now encounter the constraint $c_1$, $g(W, X) = g(X, f(a))$. To compute the solved form of $c_0 \wedge c_1$, we first use $S_0$ to eliminate $X$ and $Y$ from $c_1$. That is, we compute eliminate$(c_1, S_0)$. This gives $c_2$ which is $g(W, f(V)) = g(f(V), f(a))$. We now compute the solved form, $S_1$, of $c_2$ which is $W = f(a) \wedge V = a$. We can use $S_1$ to eliminate $V$ from $S_0$, giving $S_2$ which is $X = f(a) \wedge Y = a$. The solved form of $c_0 \wedge c_1$ is therefore simply $S_2 \wedge S_1$.

We can see that this is correct by understanding the whole process in terms of how unify operates on the constraint

$$g(X, Y) = g(f(V), V) \wedge g(W, X) = g(X, f(a)).$$

Essentially, what is happening is that we are choosing to process all of the equations originating from $c_0$ before processing the equation $c_1$. We are also delaying the elimination of variables in $c_1$ until we start processing $c_1$.

The following incremental tree constraint solving algorithm is based on this idea. We first use the current solved form $S$ to eliminate variables from the new primitive constraint $c$. Then we compute the solved form $S_1$ of $c$. If $c$ is unsatisfiable, the whole system is unsatisfiable, otherwise the new solved form is the result of conjoining $S_1$ with $S$ after using $S_1$ to eliminate the new non-parametric variables from $S$.

```
S is a global variable containing the solved form of the current tree
constraint;
c is a term equation;
and S1 is a solved form term constraint.

inc_tree_solve(c)
    c := eliminate(c, S)
    S1 := unify(c)
    if S1 ≡ false then return false endif
    S := eliminate(S, S1) ∧ S1
    return true
```

**Figure 10.11** Incremental tree solving algorithm.

**Algorithm 10.4:** Incremental tree constraint solver
INPUT: A term equation $c$.
GLOBALS: $S$ is a solved form tree constraint.
OUTPUT: Returns *true* or *false* indicating whether $S \wedge c$ is satisfiable or not. Updates the global constraint store $S$.
METHOD: The algorithm is given in Figure 10.11. □

***Example 10.7***
Let us re-examine the successful derivation for the goal `append([a],[b,c],L)` shown in Figure 6.4 in Section 6.2. The constraints that are collected in this derivation are

$$[a] = [F|R], \quad [b,c] = Y, \quad L = [F|Z], \quad R = [], \quad Y = Y', \quad Z = Y'.$$

After collecting the first two constraints, the constraint store, $S$, is

$$F = a \wedge R = [] \wedge Y = [b,c].$$

The next constraint encountered, $c$, is $L = [F|Z]$. A call to inc_tree_solve($c$) is made. First, the variables from the current constraint store $S$ are eliminated from $c$, giving $L = [a|Z]$. The call to unify returns $S_1$, the solved form of this constraint. This is simply $L = [a|Z]$ since it was already in solved form. Finally, $S_1$ is used to eliminate the non-parametric variables in $S_1$ from $S$ (but there are no occurrences) and $S$ is updated by conjoining it with $S_1$. This gives the new store

$$F = a \wedge R = [] \wedge Y = [b,c] \wedge L = [a|Z].$$

After processing $R = []$ and $Y = Y'$, the constraint store is

$$F = a \wedge R = [] \wedge Y = [b,c] \wedge L = [a|Z] \wedge Y' = [b,c].$$

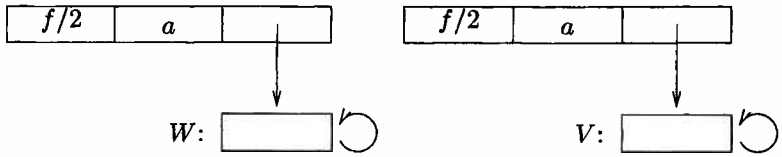
The last constraint added to the store is $Z = Y'$. After eliminating variables using the current constraint store we obtain $Z = [b,c]$. This is already in solved form. It is now used to eliminate occurrences of $Z$ from the constraint store and then

conjoined with the store, giving

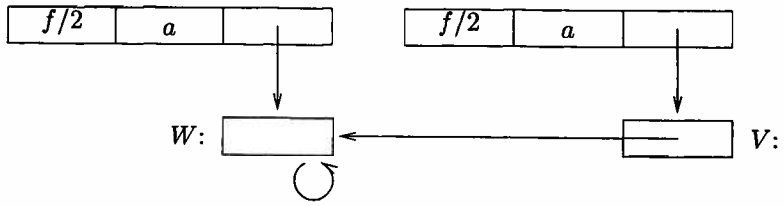
$$F = a \wedge R = [\,] \wedge Y = [b, c] \wedge L = [a, b, c] \wedge Y' = [b, c] \wedge Z = [b, c].$$

The incremental tree constraint solvers used in CLP systems do not really manipulate equations and compute solved forms as we have described here. Instead terms are represented by dynamic data structures. Variables are represented by a single memory cell which is a pointer. An unconstrained variable is represented by a self-pointer. If the variable is equated to some tree (in solved form), the variable cell is set to point to the memory cell of the root of the tree. Trees with $n$ children are represented by $n + 1$ contiguous memory cells. The first cell contains the tree constructor and the number of children, $n$, while the remaining cells hold pointers to the $n$ child trees. A common optimization is to store a sub-tree with no children in the cell rather than to use a pointer to the sub-tree.

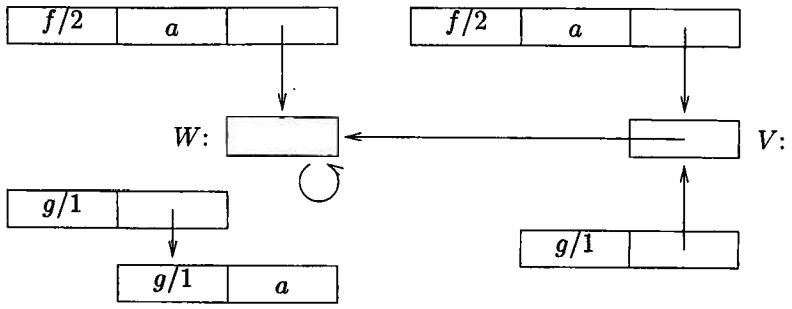
For instance, imagine we encounter the term equation $f(a, W) = f(a, V)$. The terms $f(a, W)$ and $f(a, V)$ are represented by:



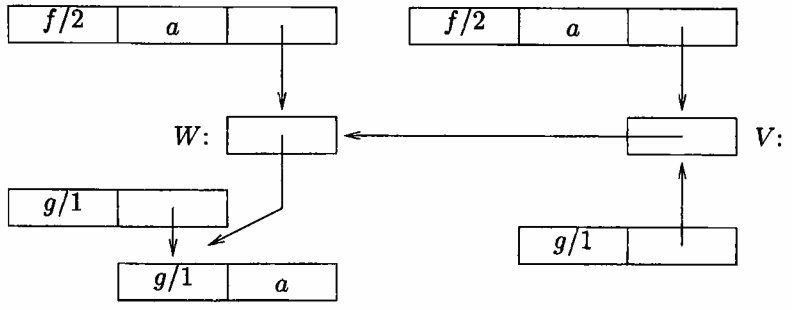

Representing terms and variables dynamically with pointers allows elimination of a variable to be very efficient, since it requires only one pointer change. When we solve $f(a, W) = f(a, V)$, the tree solving algorithm checks that the topmost tree constructors are the same and have the same number of children, and then iteratively unifies the arguments of the two terms. The two first arguments are already identical so no change happens. To equate the second arguments, the algorithm points the cell for $V$ at $W$, in effect replacing $V$ by $W$ everywhere it occurs. Note that the solving algorithm could just as well have chosen to point $W$ at $V$. This gives




which represents the solved form $V = W$. Imagine we now encounter the equation $g(g(a)) = g(V)$. We first add the cell representation of the terms $g(g(a))$ and $g(V)$. This gives:




Note that, since $V$ has only a single node representing it, elimination of $V$ from $g(V)$ has been done automatically. That is because, whenever we access the node for $V$, we will follow the link to $W$. If we now solve $g(g(a)) = g(V)$, after checking that the top most tree constructors are the same, we try to equate $g(a)$ and $V$. Following the link from $V$ to $W$, we set this to point at $g(a)$, giving



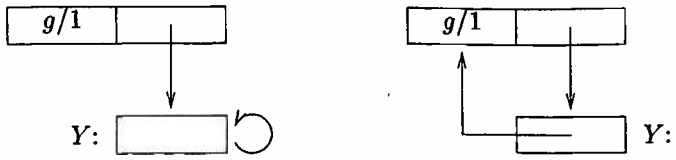

This represents the solved form $V = g(a) \wedge W = g(a)$.

Many real-world CLP systems employ an incorrect solver. The reason is that when a constraint of the form $x = t$ is encountered in the tree solver, where $x$ is a variable, it is costly to test whether $x$ occurs in $t$. Even using the pointer representation, the simple tree solving algorithm is still quadratic. For this reason CLP solvers usually omit this *occurs check*. This is, in effect, omitting case (5) in the algorithm for unify shown in Figure 1.7. This means, using the pointer representation, the solver has linear time complexity. Even more important, the most common form of unification, in which a new variable is equated to a term, requires only a single pointer to change.

Unfortunately, simplification employed in CLP systems typically removes all hints that a missing occurs check may have caused a problem. For example, in a typical system the goal p with program

```
p  :-  Y = g(Y).
```

will execute by first building the structure shown on the left below and unification will erroneously succeed with the cells in the state shown on the right.




The system will return the answer *true*. Worse still, if the solver is applied to structures which include loops, the solver may execute forever. For example, this is likely to happen with the goal q and program

```
q  :-  Y = g(Y), Z = g(Z), Z = Y.
```

In practice however, few programs require the occurs check to execute correctly and therefore incorrectness of the constraint solver is rarely an issue.

### 10.6.2 Incremental Consistency Based Solvers

In Chapter 3 we examined constraint solving methods based on repeatedly using constraints to remove inconsistent values from a variable's domain and propagating these new domains to other constraints until no new inconsistent values in the domains are found. We now examine how we can make such consistency based solvers incremental. Thankfully, this is quite simple because propagation is essentially an incremental process

We give an incremental solver based on bounds consistency. It is a straightforward task to modify it to handle other forms of consistency methods.

**Algorithm 10.5:** Incremental bounds consistency solver
INPUT: A primitive integer constraint $c$.
GLOBALS: An integer constraint $S$, together with a range domain $D$ assigning ranges to the variables in $S$ and $c$.
OUTPUT: Returns *true*, *false* or *unknown* depending on the satisfiability of $S \wedge c$ and updates the global variables.
METHOD: The algorithm is shown in Figure 10.12. □

Essentially, the incremental solver is a copy of the bounds consistency algorithm of Figure 3.13. The only difference is that when we add a new primitive constraint $c$ this is the only element of the active set $C_0$. The process begins by first applying the consistency rules of $c$, and then continuing propagation in the normal manner. Note that, if $c$ causes no propagation of information, that is, applying the propagation rules for $c$ does not change the domain of any variable, the solver terminates immediately after adding $c$. Similar incremental consistency based solvers are employed in most CLP systems that solve finite domain constraints.

One slight complication when using an incremental consistency based solver, is the need to incrementally collect domain information. As new variables are encountered during a derivation, we need to define and collect their initial domain information. For instance, evaluation of the domain declaration `X :: [4..8]` will

```
c, c1, ..., cn are primitive constraints;
S is a global variable holding the current constraint store;
D is a global variable holding the current domain;
D1 is a domain;
C0 is a set of primitive constraints;
and x is a variable.

inc_bounds_solv(c)
    S := S ∧ c
    let S be of form c1 ∧ ··· ∧ cn
    C0 := {c}
    while C0 ≠ ∅ do
        choose c ∈ C0
        C0 := C0 \ {c}
        D1 := bounds_consistent_primitive(c, D)
        if D1 is a false domain then return false endif
        for i := 1 to n do
            if there exists x ∈ vars(ci) such that D1(x) ≠ D(x) then
                C0 := C0 ∪ {ci}
            endif
        endfor
        D := D1
    endwhile
    if D is a valuation domain then
        return satisfiable(S, D)
    else return unknown
```

**Figure 10.12** An incremental finite domain propagation-based solver.

set the initial range for variable $X$. In effect, it must extend the domain $D$ to map $X$ to the set of values $D(X) = [4..8]$. If $D(X)$ is already defined, the declaration is equivalent to the constraint $4 \leq X \wedge X \leq 8$. If a variable is encountered in a finite domain constraint before it has had a domain declared, it is given a default (usually large) integer domain.

For example, executing the goal

```
X :: [4..8], Y :: [0..3], Z :: [2..2], X = Y + Z, Y ≠ Z.
```

first processes the domain declarations. By the time the first equation is processed, the global constraint store $S$ is still *true* and the global domain $D$ is given by

$$D(X) = [4..8], \quad D(Y) = [0..3], \quad D(Z) = [2..2].$$

Execution of the call to inc_bounds_solv to add $X = Y + Z$ proceeds as follows. $S$ becomes $X = Y + Z$, and $C_0$ becomes $\{X = Y + Z\}$. $X = Y + Z$ is removed from $C_0$ and the procedure bounds_consistent_primitive evaluates the propagation rules for it, updating the global domain to

$$D(X) = [4..5], \quad D(Y) = [2..3], \quad D(Z) = [2..2].$$

Because the variable $X$ has changed its range, the constraint $X = Y + Z$ is added to $C_0$. Again $X = Y + Z$ is removed from $C_0$, but, this time, its propagation rules cause no change in the domain. Since $C_0$ is empty, the **while** loop finishes and *unknown* is returned by the solver.

Next inc_bounds_solv($Y \neq Z$) is called. $S$ becomes $X = Y + Z \wedge Y \neq Z$ and $C_0$ is $\{Y \neq Z\}$. $Y \neq Z$ is removed from $C_0$ and its propagation rules determine the new domain

$$D(X) = [4..5], \quad D(Y) = [3..3], \quad D(Z) = [2..2].$$

Since the range of $Y$ has changed, both $X = Y + Z$ and $Y \neq Z$ are added to $C_0$. Removing $X = Y + Z$ from $C_0$ and applying its propagation rules modifies the domain to

$$D(X) = [5..5], \quad D(Y) = [3..3], \quad D(Z) = [2..2].$$

Since the domain of $X$ has changed, $X = Y + Z$ is put back into $C_0$. In the remainder of the computation, $X = Y + Z$ and $Y \neq Z$ are re-examined but this does not change the domain. The call terminates with answer *true*, since the domain is a valuation domain which satisfies the constraints.

Note that, although it is possible to build an incremental *complete* consistency based solver, this is unlikely to be worthwhile. This is because the backtracking search that is part of the complete consistency based solver, effectively sets each variable to a single value. Any new constraint that does not agree with this assignment can potentially cause the undoing of this search. Since the backtracking search is usually the dominant part of the computation time for the complete consistency based solver there is little incrementality, since at every step most of the work may need to be redone. Rather, as we have seen, in the CLP paradigm it is the role of the programmer to explicitly invoke a complete backtracking search at the end of computation by calling a labelling predicate.

## 10.7 (*) Incremental Real Arithmetic Solving

We conclude the chapter by investigating how to build an incremental solver for arbitrary arithmetic constraints. This algorithm incorporates techniques from Gauss-Jordan elimination, the simplex algorithm and local propagation to give an efficient incremental constraint solver for arbitrary real arithmetic constraints. This algorithm, or closely related variants, is the basis for arithmetic constraint solving in many current CLP systems.

Our first step is to develop an incremental solver for linear inequalities. A key feature of the Gauss-Jordan algorithm which allowed it to be a easily modified into an incremental algorithm, is that it processes the constraints one at a time, transforming them into a solved form by using each constraint to eliminate a variable. The simplex satisfaction algorithm also uses variable elimination to transform con-

straints into a solved form, namely basic feasible solved form. This suggests that the simplex satisfaction algorithm might be a good basis for an incremental solver for arithmetic inequalities. As we shall see, this is correct—it is straightforward to modify the simplex algorithm to give an incremental solver for linear inequalities.

For the moment, we shall assume that the linear inequality $c$ has been rewritten into an equality by adding a slack variable and that all variables are constrained to be non-negative. The constraint is to be added to the constraint store $S$, which is in basic feasible solved form. The first step is to use $S$ to eliminate basic variables from $c$. We handle the case when all variables are eliminated in the obvious way.[2]

In the case when there are variables left, processing continues by adding an artificial variable, $z$, to $c$ and then using the simplex algorithm to minimize $z$. If minimization results in an objective function with associated constant $b_1$ equal to 0, then the optimum value for $z$ is 0 and so the store remains solvable after adding the constraint. The chief difficulty is removing the artificial variable from the basic feasible solved form, $S_1$, returned from the simplex algorithm. There are two cases. The first case is when $z$ is a parameter. In this case we need only remove all occurrences of $z$. The second case is when $z$ is a basic variable. That is, it occurs in an equation of the form $z = \sum_{j=1}^{m} a_j x_j + b$. Now, because the objective function is equal to $z$ and has value 0 at the corresponding basic feasible solution, the constant $b$ must also be 0. Therefore, we can choose any variable, $x_J$ say, with a non-zero coefficient $a_J$ and pivot on this constraint to make $x_J$ basic and make $z$ a parameter. Since $b$ is 0, the resulting constraint will still be in basic feasible solved form. We can then eliminate $z$ from this solved form. In essence the algorithm does this, though pivoting and elimination are performed in a single step. Note that, since the original constraint viewed as an equality was not redundant, some $a_j$ must be non-zero.

**Algorithm 10.6:** Incremental simplex solver
INPUT: A linear arithmetic inequality $c$.
GLOBALS: A global linear arithmetic constraint $S$ in basic feasible solved form.
OUTPUT: Returns *true* or *false* indicating whether $S \wedge c$ is satisfiable or not and updates the global variable $S$.
METHOD: The algorithm is given in Figure 10.13. All variables are implicitly restricted to be non-negative. □

2. Note that if we only add equations which have been obtained from inequalities by adding distinct new slack variables, then variable elimination will always leave at least one variable—the new slack variable. However, it will prove useful to allow arbitrary equations.

```
c is a linear inequality written as an equality by means of a slack
variable;
S is the global constraint store in basic feasible solved form;
y, x_1, ..., x_m, z are non-negative real variables;
b, b_1, a_1, ..., a_m, d_0, ..., d_m are real constants;
e, f and f_1 are linear expressions;
c_1 and c_2 are linear equations;
and S_1 is a constraint in basic feasible solved form.


inc_simplex(c)
    c := eliminate(c, S)
    if c can be written in a form without variables then
        return eval(c)
    endif
    let c be of form b = Σ_{j=1}^{m} a_j x_j where b ≥ 0
    f := b − Σ_{j=1}^{m} a_j x_j
    c_1 := (z = b − Σ_{j=1}^{m} a_j x_j)
    ⟨Flag, S_1, f_1⟩ := simplex_opt(S ∧ c_1, f)
    let f_1 be of form b_1 + d_0 z + Σ_{j=1}^{n} d_j x_j
    if b_1 > 0 then return false endif
    if d_0 ≠ 0 then
        % artificial variable z is a parameter
        remove z from S_1
        S := S_1
    else
        % artificial variable z is not a parameter
        let c_2 be the constraint in S_1 in which z is basic
        let c_2 be of form z = Σ_{j=1}^{m} a'_j x_j
        remove c_2 from S_1
        choose J ∈ {1, ..., m} such that a'_J ≠ 0
        e := Σ_{j=1, j≠J}^{m} (a'_j / a'_J) x_j
        S := eliminate(S_1, x_J = −e) ∧ x_J = −e
    endif
    return true
```

**Figure 10.13** Incremental simplex.

***Example 10.8***

To understand how the incremental simplex algorithm works consider the constraints from problem ($P1$) on page 65:

$$\begin{aligned} X + Y &= 3 \ \wedge \\ -X - 3Y + 2Z + T &= 1 \ \wedge \\ X \geq 0 \wedge Y \geq 0 \wedge Z \geq 0 \wedge T &\geq 0. \end{aligned}$$

Initially the global constraint store, $S$, is set to *true*. The first call to inc_simplex adds the constraint $X + Y = 3$. This sets $c_1$ to $A = 3 - X - Y$ and $f$ to $3 - X - Y$ where $A$ is the name of the artificial variable. The function simplex_opt is called to

solve the problem

minimize $3 - X - Y$ subject to

$$\begin{aligned} A &= 3 - X - Y \ \wedge \\ X \geq 0 &\wedge Y \geq 0 \wedge A \geq 0. \end{aligned}$$

Either $X$ or $Y$ can be chosen as the entry variable since both have negative coefficients in the objective function. Suppose that $X$ is chosen, then the exit variable must be $A$. After pivoting we obtain

minimize $A$ subject to

$$\begin{aligned} X &= 3 - A - Y \ \wedge \\ X \geq 0 &\wedge Y \geq 0 \wedge A \geq 0. \end{aligned}$$

As no variable in the objective function has a negative coefficient, we have reached the optimal solution. On return from simplex_opt, $S_1$ will be $X = 3 - A - Y$ and $f_1$ will be $A$. Rewriting $f_1$ to the form $0 + 0 \times X + 0 \times Y + 1 \times A$, it is clear that $b_1$ is 0 and $d_0$ is 1. Thus, all occurrences of $A$ are removed from $S_1$, giving $X = 3 - Y$, and the global constraint store, $S$, is set to this value. The value *true* is returned indicating the store remains satisfiable.

Next inc_simplex is called with $-X - 3Y + 2Z + T = 1$. First, $c$ is set to the result of using the constraint store, $X = 3 - Y$, to eliminate variables from the input constraint. Thus, $c$ becomes $-2Y + 2Z + T = 4$. Now $c_1$ is set to $A = 4 + 2Y - 2Z - T$ and $f$ to $4 + 2Y - 2Z - T$. The function simplex_opt is called to solve the problem

minimize $4 + 2Y - 2Z - T$ subject to

$$\begin{aligned} X &= 3 - Y \ \wedge \\ A &= 4 + 2Y - 2Z - T \ \wedge \\ X \geq 0 &\wedge Y \geq 0 \wedge Z \geq 0 \wedge T \geq 0 \wedge A \geq 0. \end{aligned}$$

Either $Z$ or $T$ can be chosen as the entry variable since both have negative coefficients in the objective function. Suppose that $T$ is chosen. The exit variable must be $A$. After pivoting we obtain

minimize $A$ subject to

$$\begin{aligned} X &= 3 - Y \ \wedge \\ T &= 4 + 2Y - 2Z - A \ \wedge \\ X \geq 0 &\wedge Y \geq 0 \wedge Z \geq 0 \wedge T \geq 0 \wedge A \geq 0. \end{aligned}$$

Since no variable in the objective function has a negative coefficient, simplex_opt terminates as the optimum has been reached. On return, $S_1$ is

$$X = 3 - Y \wedge T = 4 + 2Y - 2Z - A$$

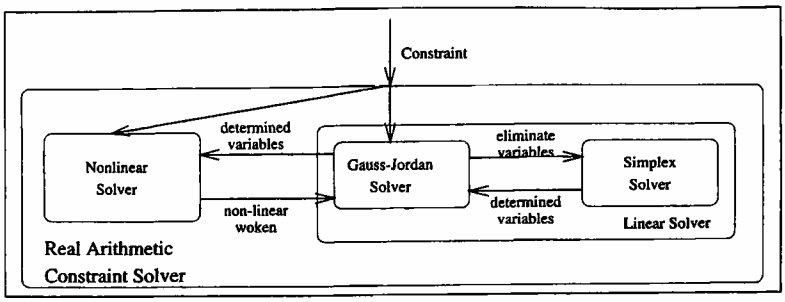



**Figure 10.14** Real arithmetic constraint solver.

and $f_1$ will be $A$. Again, $b_1$ is 0 and $d_0$ is 1. Thus, all occurrences of $A$ are removed from $S_1$, giving $X = 3 - Y \wedge T = 4 + 2Y - 2Z$, and the global constraint store, $S$, is set to this value. Finally, *true* is returned, indicating that the store remains satisfiable.

We are now in a position to build an incremental constraint solver for arbitrary arithmetic constraints. In essence, the constraint solver has three different sub-solvers: a variant of the incremental Gauss-Jordan solver for handling linear equalities, the incremental simplex algorithm for linear inequalities and a simple incomplete solver for handling nonlinears. The solver architecture is shown in Figure 10.14.

One difficulty when constructing the solver from the component solvers is that the incremental simplex algorithm cannot directly handle arbitrary linear inequalities, but only linear equations in which each variable is implicitly restricted to be non-negative. In our setting, all of the original variables appearing in the constraint have unrestricted range, it is only the slack variables introduced to change inequalities into equations which are implicitly restricted to take non-negative values.

To overcome this difficulty, the solver processes linear equations and inequalities as follows. First, if the constraint to be added, $c$, is a linear inequality, it is rewritten into linear equality by adding a slack variable. Next the solved form, $S_{GJ}$, associated with the incremental Gauss-Jordan solver is used to eliminate variables from $c$. Now if $c$ contains only slack variables, and so all variables in $c$ are implicitly restricted to be non-negative, it is sent to the incremental simplex solver. Otherwise, one of the non-slack variables in $c$ is chosen for elimination which is performed as in the incremental Gauss-Jordan solver.

What this means is that, at any point in time, the constraint

$$\mathsf{eliminate}(S_{GJ}, S_{sim}) \wedge S_{sim}$$

is essentially in basic feasible solved form where $S_{sim}$ is the solved form associated with the incremental simplex solver. The only difference to basic feasible solved form is, if a non-slack variable $x$ is basic, the constant in the equation in which $x$ is basic may not be non-negative. This is allowed since $x$ is not restricted in the values it can take.

When a primitive constraint is first sent to the solver, it is classified as linear or nonlinear and sent to the appropriate sub-solver. Linear constraints are essentially handled as described above. Nonlinear constraints are handled very simply. Each nonlinear primitive constraint, $c$, is assumed to be satisfiable until sufficient variables in $c$ are determined by $S_{GJ}$ to make $c$ linear. In this case, the now linear primitive constraint is passed to the linear equality solver. Thus the solver is complete for linear equalities and inequalities, but incomplete for nonlinear constraints.

To process nonlinear constraints efficiently we use a global variable $S_{det}$ which contains the determined solved form (see Section 1.7) associated with the fixed variables of $S_{GJ}$. Whenever a new nonlinear primitive constraint is encountered, first the variables in $S_{det}$ are eliminated. Then the primitive constraint is tested to see if it is "really" nonlinear. If it has become linear after elimination, it is processed as a linear constraint. Otherwise it is added to the set of nonlinear constraints $C_{nl}$. Whenever a constraint is added to $S_{GJ}$ or $S_{sim}$, $S_{det}$ is updated and $C_{nl}$ is checked to see if any primitive constraints have become linear. An invariant of the solver is that all variables in $S_{det}$ have been eliminated from $C_{nl}$, meaning that when $S_{det}$ is changed we need only eliminate the new variables in $S_{det}$ to $C_{nl}$. Thus in the algorithm we determine $S_\delta$, the new equalities added to $S_{det}$ because of the addition of the constraint. Then $S_\delta$ is added to $S_{det}$ and variables in $S_\delta$ are eliminated from the nonlinear constraints. Those constraints that become linear are sent to the linear constraint solver. In a sense, local propagation is used to handle nonlinear constraints.

**Algorithm 10.7:** Incremental arithmetic constraint solver
INPUT: An arithmetic constraint $c$.
GLOBALS: A global constraint store $S$ of form $S_{GJ} \wedge S_{sim} \wedge C_{nl}$ where $S_{GJ}$ is the solved form of the linear equalities associated with the incremental Gauss-Jordan algorithm, $S_{sim}$ is the solved form of the linear equalities with all slack variables and $C_{nl}$ is the set of nonlinear constraints. In addition, there is a global variable $S_{det}$ which is the determined solved form associated with the fixed variables of $S_{GJ}$.
OUTPUT: Returns *true*, *false* or *unknown* indicating whether $c$ conjoined with the global constraint store $S$ is satisfiable or not and updates the global variables.
METHOD: The algorithm is given in Figures 10.15 and 10.16. □

***Example 10.9***
To understand the operation of the incremental arithmetic constraint solver, consider the factorial program first introduced in Section 4.2:

```
fac(0, 1).
fac(N, N * F)   :-  N ≥ 1, fac(N - 1, F).
```

When evaluating the goal `N ≥ 1, fac(N,F)` the following constraints are added to the solver when constructing the first successful derivation:

$$N \geq 1, \quad N = N', \quad F = N' \times F', \quad N' \geq 1, \quad N' - 1 = 0, \quad F' = 1.$$

```
S_GJ and S_sim are conjunctions of linear equations;
C_nl is the set of nonlinear primitive constraints;
c is a primitive constraint;
x is a real variable;
e is an arithmetic expression;
and S_det and S_δ are determined solved forms.

inc_arith_solve(c)
    if c is an inequality then
        rewrite c into an equality by adding a new slack variable
    endif
    cases
    c is linear:
        if not add_linear(c) then
            return false
        endif
    c is not linear:
        if not add_nonlinear(c) then
            return false
        endif
    endcases
    if C_nl ≡ ∅ then
        return true
    else
        return unknown
    endif

add_linear(c)
    c := eliminate(c,S_GJ)
    if vars(c) ≡ ∅ then
        return eval(c)
    else if there is some x ∈ vars(c)
        which is not a slack variable then
        write c in the form x = e where e does not involve x
        S_GJ := eliminate(S_GJ, x = e) ∧ x = e
    else % c contains only slack variables
        if not inc_simplex(c) then
            return false
        else
            S_δ := new_determined_sf(S_sim)
            S_GJ := eliminate(S_GJ, S_δ)
        endif
    endif
    S_δ := new_determined_sf(S_GJ)
    S_det := S_δ ∧ S_det
    return check_nonlinears(S_δ)
```

**Figure 10.15** Incremental arithmetic constraint solving algorithm (part 1).

```
C_nl is the set of nonlinear primitive constraints;
c is a primitive constraint;
x_1, ..., x_n are real variables;
e_1, ..., e_n are arithmetic expressions;
S, S_det and S_δ are determined solved forms;
and N_tmp and W are sets of nonlinear primitive constraints.

add_nonlinear(c)
    c := eliminate(c, S_det)
    if c is not linear then
        C_nl := C_nl ∪ {c}
        return true
    else
        return add_linear(c)
    endif

new_determined_sf(S)
    S_δ := true
    let S be of the form x_1 = e_1 ∧ ··· ∧ x_n = e_n
    for i := 1 to n do
        if vars(e_i) ≡ ∅ and (x_i = e_i) not in S_det then
            S_δ := S_δ ∧ x_i = e_i
        endif
    endfor
    return S_δ

check_nonlinears(S_δ)
    N_tmp := ∅
    W := ∅
    for each c ∈ C_nl do
        c := eliminate(c, S_δ)
        if c is linear then
            W := W ∪ {c}
        else
            N_tmp := N_tmp ∪ {c}
        endif
    endfor
    C_nl := N_tmp
    for each c ∈ W do
        if not add_linear(c) then
            return false
        endif
    endfor
    return true
```

**Figure 10.16** Incremental arithmetic constraint solving algorithm (part 2).

We now describe how the incremental arithmetic constraint solver will process these constraints. Initially $S_{GJ}$, $S_{sim}$ and $S_{det}$ are each set to *true* and $C_{nl}$ is set to $\emptyset$. When $N \geq 1$ is added, it is first rewritten to $N - S_1 = 1$ where $S_1$ is a slack

variable. As this equation is linear, it is processed with add_linear. Since $S_{GJ}$ is *true*, eliminating variables from it using $S_{GJ}$ does not change it. As $N - S_1 = 1$ has a non-slack variable in it, namely $N$, it is processed by eliminating this variable. This sets $S_{GJ}$ to $N = 1 + S_1$. Next new_determined_sf is called. Since there are no fixed variables, $S_\delta$ is set to *true*. Finally, check_nonlinears is called. This does very little since there are no fixed variables and no nonlinear constraints. The constraint solver returns *true*.

Now $N = N'$ is added. Again, since this equation is linear, it is processed with add_linear. Eliminating variables from it using $S_{GJ}$ gives $1 + S_1 = N'$. As $1 + S_1 = N'$ has a non-slack variable in it, namely $N'$, it is processed by eliminating this variable. This sets $S_{GJ}$ to $N = 1 + S_1 \wedge N' = 1 + S_1$. Finally, new_determined_sf and check_nonlinears are called. Again, they do very little since there are no fixed variables and no nonlinear constraints. The constraint solver returns *true*.

Next $F = N' \times F'$ is encountered. This is nonlinear, so add_nonlinear is called. Eliminating variables in $S_{det}$, since $S_{det}$ is *true*, does not make $F = N' \times F'$ linear, and so the constraint is added to $C_{nl}$. This time, the constraint solver returns *unknown* as there are now nonlinears.

Now $N' \geq 1$ is added. It is first rewritten to $N' - S_2 = 1$ where $S_2$ is a slack variable. As this equation is linear, it is processed with add_linear. Eliminating variables from it using $S_{GJ}$ gives $1 + S_1 - S_2 = 1$. Since all variables in this equation are slack, it is processed by calling inc_simplex. After some computation this will set $S_{sim}$ to $S_1 = S_2$. Finally, new_determined_sf and check_nonlinears are called. They do very little since there are no fixed variables in either $S_{sim}$ or $S_{GJ}$. Again the constraint solver returns *unknown*.

Next $N' - 1 = 0$ is added. As it is linear, it is processed with add_linear. Eliminating variables from it using $S_{GJ}$ gives $S_1 = 0$. This involves only slack variables so inc_simplex is called. This sets $S_{sim}$ to $S_1 = 0 \wedge S_2 = 0$. Next new_determined_sf($S_{sim}$) is called, returning $S_1 = 0 \wedge S_2 = 0$. Using this to eliminate variables from $S_{GJ}$ gives $N = 1 \wedge N' = 1$. Now new_determined_sf($S_{GJ}$) is called, returning $N = 1 \wedge N' = 1$. Both $S_{det}$ and $S_\delta$ are set to $N = 1 \wedge N' = 1$ and check_nonlinears is called. Using $S_\delta$ to eliminate variables from $F = N' \times F'$ gives the linear constraint $F = 1 \times F'$, so the constraint is removed from $C_{nl}$ and processed with a recursive call to add_linear. It is unchanged by eliminating variables from it using $S_{GJ}$. As it has non-slack variables, namely $F$ and $F'$, one of these is arbitrarily chosen for elimination, say $F$. This sets $S_{GJ}$ to $N = 1 \wedge N' = 1 \wedge F = F'$. Now, new_determined_sf and check_nonlinears are called. This time they change nothing since there are no new fixed variables and no nonlinear constraints. After returning from the nested calls, the constraint solver will return *true* as now there are no nonlinears.

Note how, after adding the constraint $S_1 = 0$ to $S_{sim}$, we needed to eliminate $S_1$ and $S_2$ from $S_{GJ}$ to discover that $N$ and $N'$ are now fixed.

Finally the constraint $F' = 1$ is added. This proceeds in the obvious way. As it is linear, it is processed by add_linear. Elimination leaves it unchanged, and the

non-slack variable $F'$ is chosen for elimination. This sets $S_{GJ}$ to

$$N = 1 \wedge N' = 1 \wedge F = 1 \wedge F' = 1.$$

Now new_determined_sf is called, setting $S_\delta$ to $F = 1 \wedge F' = 1$. Then $S_{det}$ is set to $N = 1 \wedge N' = 1 \wedge F = 1 \wedge F' = 1$ and check_nonlinears is called. As there are no nonlinear constraints this has no effect. The constraint solver returns *true*.

## 10.8 Summary

In this chapter we have considered the implementation of CLP languages. This can be viewed as a variant of the backtracking algorithm given in Chapter 3 for solving CSPs and essentially performs a depth-first search through a goal's derivation tree.

One of the keys to the efficient implementation of CLP languages is the use of incremental constraint solvers. Incremental constraint solvers efficiently answer the sequence of satisfaction problems that arise in a CLP derivation. We gave incremental algorithms for Gauss-Jordan elimination, tree constraint solving and bounds consistency.

A consequence of incremental constraint solving is the need to save a global constraint store at a choicepoint and to restore it when backtracking to that choicepoint. This can be done efficiently by using time stamping and trailing. Another technique is semantic backtracking.

Constraint logic programs may also contain constructs for minimization, if-then-else, finding the first solution and negation. We showed how the last three of these constructs can be implemented using the cut, a low level search construct which prunes branches from the derivation tree. Minimization can be implemented in a number of ways. We gave two possible extensions to the backtracking goal solver for supporting minimization.

## 10.9 Exercises

**10.1.** Execute the goal `delete([a,b], X, R)` using simple_backtrack and inc_backtrack to find the first solution. Compare the number of steps in the **while** loop of unify. Recall that the predicate `delete` is defined by

```
delete([X | Xs], X, Xs).
delete([X | Xs], Y, [X | R]) :- delete(Xs, Y, R).
```

**10.2.** (*) Give an incremental algorithm for local propagation.

**10.3.** Given the program

```
p(N, P, P) :- N = 0.
p(N, P, Q) :- p(N-1, 2*P + Q, Q - P).
```

find the first answer to the goal `p(2,X,Y)` using inc_backtrack and trailing. When choosing a variable to eliminate from an equation, always select the oldest, that is to say, the variable first introduced into the store.
Repeat this evaluation but now use semantic backtracking. Compare the amount of information placed on the backtrack stack.
Perform the comparison again, but this time choose to eliminate the youngest variable from each equation.

**10.4.** Give an algorithm for semantic backtracking based on the example in Section 10.3. Illustrate its use with the program from Example 10.3 and the goal `sum(1,S), S=2`.

**10.5.** Execute the goals `max(4, 3, M)` and `max(3, 4, M)` using inc_backtrack and the program:

```
max(X, Y, X) :- X ≥ Y, !.
max(X, Y, Y).
```

The predicate `max(X,Y,Z)` is meant to hold if $Z$ is the maximum of $X$ and $Y$. Now try the goal `max(4, 3, 3)`. Is this the answer you expected? Modify the program (without using if-then-else) so that the expected answer occurs.

**10.6.** (*) Evaluate the execution of the goal

```
L = dl([a,b|T], T), dl_append(L, L, R)
```

for the program

```
dl_append(dl(H1,T1), dl(H2,T2), dl(H3,T3)) :-
     H3 = T3, T1 = H1, T3 = T2.
```

with and without the occurs check.

**10.7.** Give an example showing that the tree constraint solver without the *occurs check* is not well-behaved.

**10.8.** (*) Does the incremental arithmetic solver find all fixed variables? [Hint: consider what happens if the constraints $X \geq 2$ and $X \leq 2$ are added to the solver. Consider how the incremental simplex algorithm could be modified to find more fixed variables.]

**10.9.** (*) Extend the incremental arithmetic solver to handle disequations, that is arithmetic constraints of the form $s \neq t$, by delaying evaluation of such constraints until all variables in $s$ and $t$ are fixed. Explain how you can now handle strict inequalities, that is arithmetic constraints of the form $s < t$ and $s > t$. Is the solver complete? [Hint: consider your answer to the above question.]

## 10.10 Notes

Implementation techniques for CLP languages are based on those for Prolog. In particular, most Prolog systems use the same basic evaluation strategy and trail information to allow backtracking. The constraint solving algorithm in Prolog is an incremental tree constraint solver. The efficient implementation using pointers was first used in the WAM [142, 5].

The incremental arithmetic constraint solving algorithm given here is a variant of the algorithm used in one of the earliest CLP languages, $CLP(\mathcal{R})$ [75]. Similar algorithms are used in other CLP languages and constraint solving toolkits. The earliest CLP languages used time stamping and trailing to efficient implement backtracking, see for example $CLP(\mathcal{R})$ [75]. The term *semantic backtracking* is due to Van Hentenryck and Ramachandran who use it in an implementation of $CLP(\Re_{Lin})$ [137]. They also compare it to time stamping and trailing. Recent CLP systems use a revised simplex method and semantic backtracking to implement a backtrackable incremental linear arithmetic solver (see for example [110]).

One important issue arising in the implementation of linear arithmetic constraint solvers is representation of numbers. Two common approaches are infinite precision rationals and (double-precision) floating point numbers. The advantage of rationals is that there is no possibility of numerical inaccuracy. Since decisions about constraint satisfiability may be in error because of inaccuracies caused by floating point roundoff, the problem of precision is important, as it may be completely invisible to the user if the solver erroneously answers $false$ for a satisfiable constraint.

Unfortunately, rational numbers are difficult to implement efficiently and can become too large for efficient computation even for small programs and goals. Nor do they allow the use of trigonometric functions without introducing the possibility of inaccuracy in representation. This means that rational number based systems cannot handle some kinds of programs (for example, the mortgage program of Section 5.3). An alternative approach is to use finite domains with floating point boundaries to represent ranges of real numbers. This was suggested by Cleary [31] and Hyvönen [67]. For more discussion see the notes at the end of Chapter 3. Its first use in CLP languages was in BNR Prolog discussed in [98, 99]. There is also some overhead in this method, but it has the advantage of being safe and reasonably efficient.

In this book we defined the constructs for negation, if-then-else and once directly, and then showed how they can be implemented using the cut. Usually the cut is introduced first and these constructs are defined in terms of it. However, almost all uses of the cut can be replaced by one of these three constructs, and since they are conceptually simpler than the cut, we chose to introduce them first. The cut was included in the first Prolog implementation and gets its name from first-order resolution.