7 Controlling Search

In the preceding chapters we have investigated how a programmer can model a constraint problem by using a CLP program. However constructing an initial model is often only the first part of the constraint programmer's task. In many cases, execution of the initial model may be too inefficient to solve the original problem and, in the extreme case, may not terminate. In such cases, a second phase of programming is required in which the programmer modifies the original program in order to improve its efficiency.

The size and shape of the derivation tree (also called the search space) is the primary factor in determining the efficiency of a CLP goal. If the derivation tree has an infinite derivation then the goal may not terminate, or if it is too large and the answers are not near the left part of the derivation tree, it may take too long to find a solution. In order to write efficient CLP programs the programmer therefore needs to understand the underlying search strategy and solver and the way they shape a goal's derivation tree.

In this chapter we look at how we can measure the efficiency of a CLP program. We describe several basic programming techniques—rule reordering, reordering of literals in a rule body and addition of extra constraints—which can be used to improve the efficiency of a CLP program by reducing its search space. We also describe standard CLP built-ins—if-then-else and once—which allow the programmer to tell the system not to search part of the derivation tree.

7.1 Estimating the Efficiency of a CLP Program

In Chapter 4 we saw that a CLP system evaluates a goal by searching the derivation tree for the goal. Reducing the size of the derivation tree is the main and most important strategy for improving the efficiency of CLP programs. The size and shape of a goal's derivation tree depends upon the order of literals in the program and goal and also upon the constraint solver. The order in which branches in the tree are searched depends on the order in which the rules for a predicate appear in the program. This means that when writing constraint logic programs the programmer needs to be aware that literals are evaluated left-to-right in goals and that rules are tried in textual order, as ignoring this may lead to inefficient programs, and even to that most inefficient of programs—one that is non-terminating. Conversely, using knowledge about the evaluation and the independence of the answers from literal ordering or rule ordering (discussed in Section 4.8) can allow a programmer to dramatically improve the performance of their programs.

In order to reason about the efficiency of a program, the programmer requires some way to estimate the cost of evaluating a goal. Unlike most programming languages, CLP languages are *non-deterministic*, that is, a goal can have more than one answer. As we have seen, their fundamental evaluation strategy is to explore the derivation tree from a goal in a depth-first manner. The overwhelming cost in goal evaluation is this search through the derivation tree to find the answers. Thus, the size of a goal's derivation tree (that is the number of states in the tree) is a rough measure of the cost of evaluating the goal. Although this measure does not take into account the cost of constraint solving or that we may only want to find the first answer, not all answers, it provides a rough guide when developing a program.

Another difference between CLP languages and most other languages is that CLP programs are very versatile and may be used to answer a wide variety of different queries. Therefore, when analyzing the efficiency of a program, it is important to consider the intended *mode of usage* of the user-defined constraints occurring in the program.

Definition 7.1
A *mode of usage* for a predicate p is a description of the way in which the arguments of calls to p will be constrained at the time when calls to p are evaluated.

Usually we shall specify a predicate's mode of usage by detailing which arguments to the predicate are *fixed* and which are *free*. An argument x is fixed in a particular call if, at the time of evaluation, the constraint store implies x can only take a single value. An argument x is free in a particular call if, at the time of evaluation, x can take any value in the constraint domain and still satisfy the constraint store. In the case of tree constraints we shall sometimes describe the type of term a variable is constrained to equal, for example a list or a list of fixed size. Other modes of usage are also possible, such as the argument is bounded above or the argument is bounded below.

Definition 7.2
A goal G *satisfies* a mode of usage for predicate p if, for every state in the derivation tree for G of form

$$\langle p(s_1, \ldots, s_n), L_2, \ldots, L_m \mid C \rangle,$$

the effect of the constraint store C on the arguments s_1, ..., s_n of p is correctly described by the mode of usage.

To make this discussion a little more concrete, consider the following program which sums the numbers in a list:

```
sumlist([], 0).
sumlist([N|L], N + S)  :-  sumlist(L, S).
```

One reasonable mode of usage for `sumlist` is that the first argument is fixed and that the second argument is free. The goals `sumlist([1], S)` and

```
L=[1,2], S > Z, sumlist(L,S).
```

both satisfy this mode of usage while the goals `sumlist(L, 2)` and

```
S > 3, sumlist(L,S), L=[1,2].
```

do not. To see that the goal `sumlist([1], S)` satisfies this mode of usage, consider Figure 7.1 which shows the derivation tree for this goal. The predicate `sumlist` is called in the states

$$\langle sumlist([1], S) \mid true\rangle,$$
$$\langle sumlist(L', S') \mid [1] = [N'|L'] \land S = N' + S'\rangle.$$

It is clear that in the first call, *sumlist*$([1], S)$, the first argument is fixed and the second is free. This is also true for the second call, *sumlist*(L', S'), since the associated constraint store constrains L' to be $[]$ while S' is still free to take any value (although S will be forced to be $1 + S'$).

An *intended mode of usage* for a program specifies a class of goals for which the program is designed. Any goal which satisfies the intended mode of usage is part of this class.

Efficiency of a program or predicate depends on its intended mode of usage. For a given mode of usage, the efficiency of predicate p is a measure of the size of the derivation trees for any state of the form

$$\langle p(s_1, \ldots, s_n) \mid C\rangle$$

for which the intended mode of usage describes the effect of C on the arguments $s_1, \ldots, s_n$ of p.

For example, consider the efficiency of the `sumlist` program for the mode of usage in which the first argument is a list of fixed length. We must consider how the size of the derivation tree for `sumlist(L,S)` varies with the length of the list L. For the goal `sumlist([],S)`, the size of the derivation tree is 6. For a goal with a list of length 1, we can see (from Figure 7.1) that the derivation tree has size $5 + 6 = 11$. The case when the first argument is a list with two elements gives a derivation tree of size $5 + 11 = 16$ and, in general, if the list has n elements, the derivation tree has size $5n + 6$. Thus, for this mode of usage, the size of the derivation tree is linear in the size of the initial goal.

As another example, consider `sumlist` for the mode of usage in which the first argument is free and the second argument is fixed. In this case the derivation tree is infinite, indicating that this is probably not a suitable mode of usage for this program.

Usually a program is designed to be as efficient as possible for one or two intended modes of usage. The programmer should, of course, add comments to the program to detail these modes. Standard practice is to add a procedure preface which explicitly

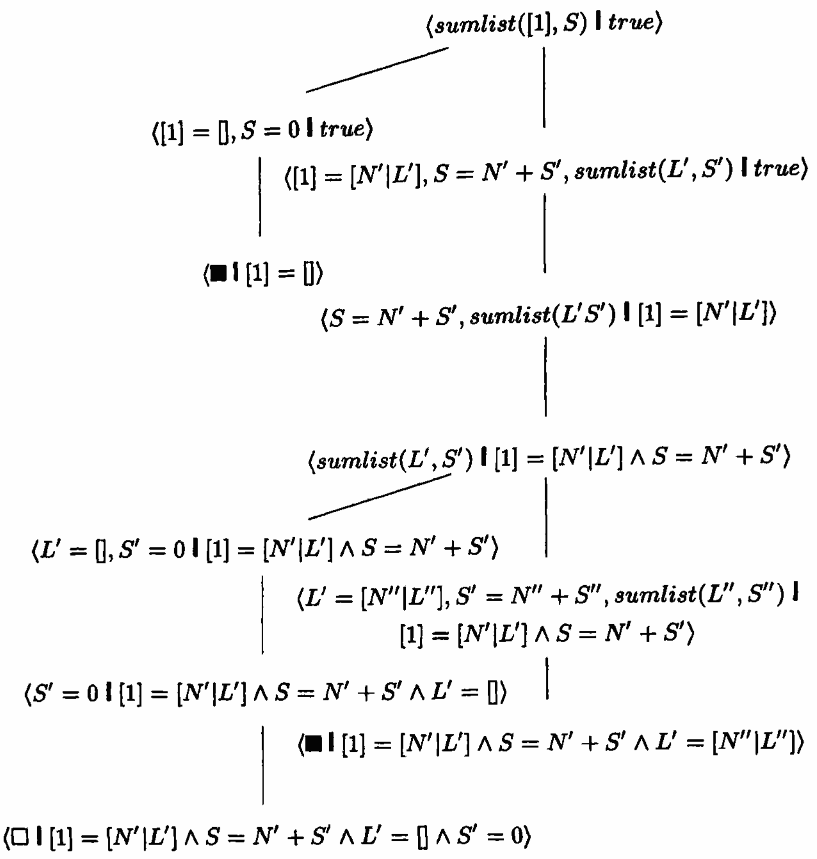

$\langle sumlist([1], S) \mid true \rangle$
$\langle [1] = [], S = 0 \mid true \rangle$
$\langle [1] = [N'|L'], S = N' + S', sumlist(L', S') \mid true \rangle$
$\langle \blacksquare \mid [1] = [] \rangle$
$\langle S = N' + S', sumlist(L'S') \mid [1] = [N'|L'] \rangle$
$\langle sumlist(L', S') \mid [1] = [N'|L'] \wedge S = N' + S' \rangle$
$\langle L' = [], S' = 0 \mid [1] = [N'|L'] \wedge S = N' + S' \rangle$
$\langle L' = [N''|L''], S' = N'' + S'', sumlist(L'', S'') \mid [1] = [N'|L'] \wedge S = N' + S' \rangle$
$\langle S' = 0 \mid [1] = [N'|L'] \wedge S = N' + S' \wedge L' = [] \rangle$
$\langle \blacksquare \mid [1] = [N'|L'] \wedge S = N' + S' \wedge L' = [N''|L''] \rangle$
$\langle \square \mid [1] = [N'|L'] \wedge S = N' + S' \wedge L' = [] \wedge S' = 0 \rangle$

Figure 7.1 Derivation tree for `sumlist([1],S)`.

details the intended mode and to order the arguments so those arguments which are intended to be fixed are placed first and those arguments which are intended to be free are placed last. Thus the `sumlist` program should be written as:

```
% sumlist(L,S): S is the sum of the elements in the fixed size list L
sumlist([], 0).
sumlist([N|L], N + S) :- sumlist(L, S).
```

where `%` indicates that the rest of the line is a comment. In general, we will not include comments in our example programs since the surrounding text will describe the program and its intended modes of usage, but good CLP programmers will, naturally, include comments as part of their programs.

The reader should be aware, however, that the size of the derivation tree is only a coarse guide to program efficiency. Sometimes execution will only search the derivation tree until the first solution is found, in which case only the size of this part of the derivation tree is important. Furthermore, derivation tree size is not a good measure for comparing the efficiency of programs whose derivation trees are of the same order of magnitude in size. In this case, since a detailed theoretical comparison requires extensive knowledge of the underlying solver and its performance characteristics, empirical comparison of the programs is usually required. Because the derivation tree measure is coarse, we can also use the size of the simplified derivation tree as a measure of program efficiency. In this case we, in effect, count only the number of branches in the derivation tree and ignore constraint collection steps.

We now look at an example illustrating several techniques which may be used to improve the efficiency of a program.

7.2 Controlling Search: An Example

Imagine that we have been asked to write a program to compute

$$S = 0 + 1 + 2 + \cdots + N$$

for a given N. That is to say, we must write a program defining `sum(N,S)` which constrains N and S so that $S = 0 + 1 + 2 + \cdots + N$ and we want the program to work efficiently for goals of the form `sum(N,S)` where N is some fixed integer. This doesn't seem too difficult. We can reason recursively: the sum of the first N numbers is the sum S of the first $N - 1$ numbers plus N, and the sum of the first 0 numbers is 0. Our first attempt might be the following simple program:

```
sum(N, S + N)  :-  sum(N - 1, S).   (S1)
sum(0, 0).                          (S2)
```

Now consider what will happen if we type the goal `sum(1,S)` and ask our CLP system to evaluate it. We wait for a little while, then wait some more. Finally our system tells us that it has run out of stack space. What has gone wrong? To understand we need to look at the simplified derivation tree for the goal shown in Figure 7.2.

The problem is that the leftmost derivation in the tree is infinite. With a depth-first left-to-right search this is the first derivation which will be explored and, since the derivation is infinite, the system never gets a chance to find the successful derivation which is to the right.

After a little thought, we realise that we could reorder the two rules. In this way the successful derivation will be encountered first. The revised program is

```
sum(0, 0).                          (S3)
sum(N, N + S)  :-  sum(N - 1, S).   (S4)
```

Now when we type the goal `sum(1,S)`, our system (quickly) informs us that the answer is $S = 1$.

Being experienced programmers, we decide that a little more testing is required. We type the goal `sum(1,0)`. This is a new mode of usage in which both arguments

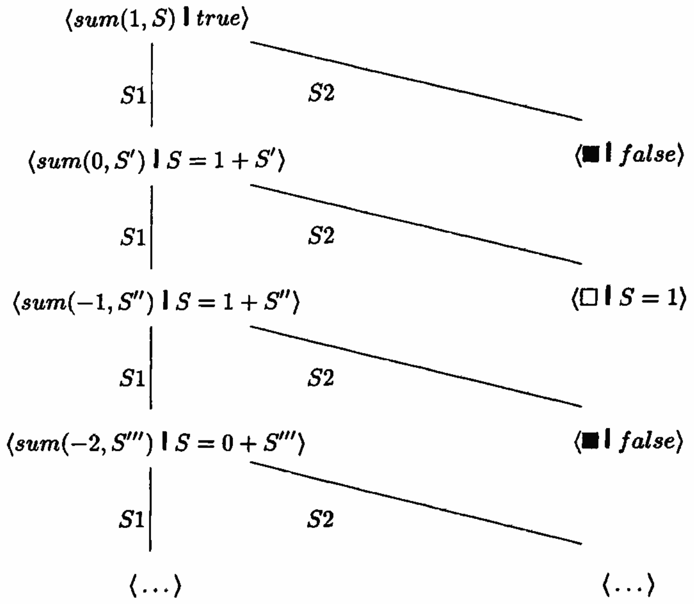

⟨sum(1, S) ❘ true⟩
S1
S2
⟨sum(0, S′) ❘ S = 1 + S′⟩
⟨■ ❘ false⟩
S1
S2
⟨sum(−1, S″) ❘ S = 1 + S″⟩
⟨□ ❘ S = 1⟩
S1
S2
⟨sum(−2, S‴) ❘ S = 0 + S‴⟩
⟨■ ❘ false⟩
S1
S2
⟨...⟩
⟨...⟩

Figure 7.2 Simplified derivation tree for `sum(1,S)`.

are fixed. Unfortunately, instead of getting the desired answer of *no*, our system again complains that it has run out of stack space. What has gone wrong? The problem is that the goal `sum(1,0)` has an infinite derivation resulting from repeatedly choosing the second rule.

$$\begin{array}{c}
\langle sum(1,0) \mid true\rangle \\
\Downarrow S4 \\
\langle sum(0,-1) \mid true\rangle \\
\Downarrow S4 \\
\langle sum(-1,-1) \mid true\rangle \\
\Downarrow S4 \\
\langle sum(-2,0) \mid true\rangle \\
\Downarrow S4 \\
\langle sum(-3,2) \mid true\rangle \\
\Downarrow \\
\vdots
\end{array}$$

How can we get rid of this unwanted derivation? Examining the simplified derivation reveals the problem. Our program was not intended to work when the first argument was negative. Therefore, the second rule is too general—it should only be used if N is greater than or equal to 1. If N is less than 1 it should fail. Thus we wish to add the constraint $N \geq 1$ to the second rule so as to prune the infinite derivation from the derivation tree. Our new program is now:

```
sum(0, 0).                                          (S5)
sum(N, N + S)   :-   sum(N - 1, S), N ≥ 1.          (S6)
```

The constraint $N \geq 1$ is redundant in the sense that any solution to the original program for sum will satisfy the constraint. But it makes this information explicit, allowing it to be used by the constraint solver earlier in the derivation tree to cause failure and so possibly reducing the size of the derivation tree.

We retry the goal `sum(1,0)`. Unfortunately, again we get a message informing us that the system has run out of stack space. After scratching our heads, we examine the derivation that always uses the second rule and see what the problem is.

$$\begin{array}{c}
\langle sum(1,0) \mid true \rangle \\
\Downarrow S6 \\
\langle sum(0,-1), 0 \geq 1 \mid true \rangle \\
\Downarrow S6 \\
\langle sum(-1,-1), -1 \geq 1, 0 \geq 1 \mid true \rangle \\
\Downarrow S6 \\
\langle sum(-2,0), -2 \geq 1, -1 \geq 1, 0 \geq 1 \mid true \rangle \\
\Downarrow S6 \\
\langle sum(-3,2), -3 \geq 1, -2 \geq 1, -1 \geq 1, 0 \geq 1 \mid true \rangle \\
\Downarrow \\
\vdots
\end{array}$$

Because of the left-to-right processing of a goal, the constraints produced by the test we have just added are never reached, so that the infinite derivation still remains. To overcome this problem we need to reorder the literals in the second rule. Our revised program is

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

We now test our program. As desired it returns *no* to the goal `sum(1,0)` and $S = 1$ to the goal `sum(1,S)`. We have now written a program which meets the original specification since the derivation tree for `sum(N,S)` is finite and linear in the size of N for the mode of usage in which N is fixed.

It is worth recapitulating the three techniques which we used in the above example to make our program terminate. First, we used rule reordering. In general, this allows us to reorder finite derivations before infinite derivations. Second, we added a constraint to a rule. This allows us to prune derivations, infinite and otherwise, from the derivation tree. Third, we reordered the literals in a rule. This allows us to move constraints so that derivations are pruned earlier. These programming techniques are vital weapons in the armoury of the constraint programmer. We will now consider their use in more detail.

7.3 Rule Ordering

The guidelines for ordering the rules in a CLP program are quite simple. The most important is that base cases, that is rules without a (possibly indirect) recursive call, should be placed first in the definition of a recursive predicate. This helps to ensure termination since it means they will be tried before the recursive call. This guideline was exemplified in the `sum` example of the previous section in which the base case `sum(0,0)` had to be placed before the recursive rule.

The second guideline is that the rules should be ordered in order of their likelihood of leading to success. This means that on average the answers will be found in the leftmost part of the derivation tree. This rule is more of a heuristic and should be verified by empirical testing.

7.4 Literal Ordering

One of the most important questions facing the novice CLP programmer is how to order the literals in a rule body. Choosing the wrong order can lead to unnecessary search and, as we saw above, in the case of a recursive program, it can lead to non-termination. There are, however, a number of simple guidelines the programmer can use. All of them depend on the intended mode of usage.

The first guideline concerns primitive constraints. These should be placed at the first place in the rule body at which they might cause failure, but no earlier than this. Constraints which cannot cause failure should be put at the end of the rule. Placing the constraint at the first point at which it might cause failure ensures that failure is determined as early as possible, thus reducing the size of the derivation tree. By leaving the addition of constraints until they might cause failure, constraints are not added unnecessarily to the constraint store. This is good as it reduces the number of constraints in the store with the result the solver uses less space and may also run faster.

In the `sum` example we saw that we needed to reorder the body

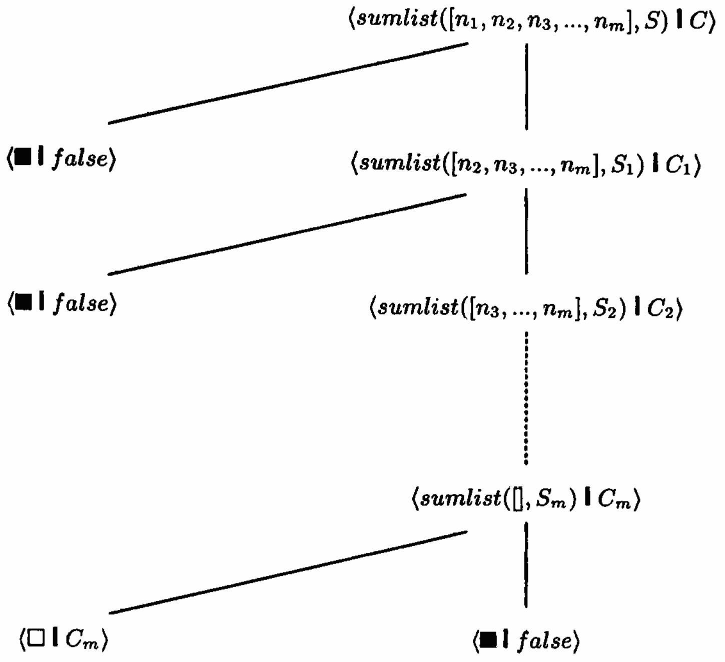
```
sum(N-1,S), N ≥ 1.
```

to

```
N ≥ 1, sum(N-1,S).
```

since the constraint `N ≥ 1` can cause failure immediately for the intended mode of usage in which N is fixed.

The remaining guidelines concern user-defined constraints. These guidelines are more heuristic in nature, and are designed to ensure failure occurs as early as possible and that choices are left until as late as possible in the computation. This will tend to increase the work done before a choice is made, meaning that work is not duplicated in different derivations.



⟨sumlist([n1, n2, n3, ..., nm], S) | C⟩
⟨■ | false⟩
⟨sumlist([n2, n3, ..., nm], S1) | C1⟩
⟨■ | false⟩
⟨sumlist([n3, ..., nm], S2) | C2⟩
⟨sumlist([], Sm) | Cm⟩
⟨□ | Cm⟩
⟨■ | false⟩

Figure 7.3 Simplified derivation tree for a call to `sumlist` with fixed first argument.

One useful guideline is to call "deterministic" predicates before "non-deterministic" predicates. A predicate is *deterministic* if it does not employ "deep" backtracking. That is, whenever a predicate definition contains multiple rules, all except one of those rules quickly leads to failure. More precisely:

Definition 7.3
A simplified derivation tree is *deterministic* if it is finite and each node in the tree has at most one child which is not a fail state.
A predicate p is *deterministic* for a particular mode of usage if, for any state of the form $\langle p(s_1, \ldots, s_n) \mid C\rangle$ satisfying that mode of usage, the states' simplified derivation tree is deterministic.

As an example, the predicate `sumlist` is deterministic for the mode of usage in which the first argument is fixed. To see this consider the simplified derivation tree shown in Figure 7.3 for a call to `sumlist` in which the first argument is a fixed list $[n_1, ..., n_m]$. Inspection of the tree reveals that it is finite and that each node has at most one child which is not a fail state. Thus `sumlist` is deterministic for this mode of usage.

The initial definition of the predicate `sum` using rules $S1$ and $S2$ is not deterministic for the mode of usage in which the first argument is fixed. This is because the simplified derivation tree in Figure 7.2 is not finite and also the node at the second choice point has two children which are not fail states. Similarly neither the reordered program ($S3$ and $S4$), nor the next modified version ($S5$ and $S6$) is deterministic. However, the final version of the program `sum` ($S7$ and $S8$) is deterministic for this mode of usage.

Predicates which are used in a mode for which they are deterministic require little search to find an answer. It also follows from the definition that they have at most one answer. Therefore, they should be called before non-deterministic predicates. As an example, recall (part of) the genealogical database from Chapter 5:

```
father(jim,edward).          mother(maggy,fi).
father(jim,maggy).           mother(fi,lillian).
father(edward,peter).
father(edward,helen).
father(edward,kitty).
father(bill,fi).
```

The predicates `father` and `mother` are deterministic for the mode of usage in which the second argument is fixed since each person has exactly one father and mother. Note that this is not the case if the first argument is fixed since people can have more than one child.

Now imagine that we wish to define the `grandfather(Z,X)` relationship which holds if Z is the grandfather of X. One way to define it is:

```
grandfather(Z,X) :- father(Z,Y), father(Y,X).
grandfather(Z,X) :- father(Z,Y), mother(Y,X).
```

We wish to use the program to efficiently answer queries of the form "Who is the grandfather Z of a given person X?" That is, the intended mode of usage is that second argument of `grandfather` is fixed and the first is free. This means that in both the first and second rules, the first literal is non-deterministic since both of its arguments are free and the last literal is deterministic—for a fixed X there is only one mother or one father. Thus for increased efficiency, we may wish to swap the two literals in each of the rules since the second literal remains deterministic and this places the deterministic call before the non-deterministic. This gives the program:

```
grandfather(Z,X) :- father(Y,X), father(Z,Y).
grandfather(Z,X) :- mother(Y,X), father(Z,Y).
```

This program is much more efficient for the intended mode of usage than the original program, since reordering has made all literals deterministic. For instance, with the original program the goal `grandfather(X,peter)` generates a simplified derivation tree with 63 states, while, with the second program, the simplified derivation tree has only 23 states. The reader is encouraged to draw both trees.

However when using this guideline it is important to remember that moving a deterministic call too early may mean that it becomes non-deterministic. In general, it should only be moved to the first point at which it becomes deterministic. For instance, in the rule

```
grandfather(Z,X) :- father(Y,X), father(Z,Y).
```

The second user-defined constraint `father(Z,Y)` is deterministic at this point since, by the time it is reached, Y has a fixed value. But, we cannot move the user-defined constraint any earlier because it must follow the call `father(Y,X)` to be deterministic.

Generally, it is a good idea to place literals with a small number of answers before those with many answers, since this will reduce the size of the search tree. In particular, literals with a large number of answers should be placed last.

Imagine that we have already defined the predicate `parent(Y,X)` which holds if Y is the parent of X as follows:

```
parent(Y,X) :- father(Y,X).
parent(Y,X) :- mother(Y,X).
```

We can define the grandfather relationship by

```
grandfather(Z,X) :- father(Z,Y), parent(Y,X).
```

However for the mode of usage in which X is fixed, we are better off to reorder the two literals in the body, since `parent(Y,X)` has at most two answers for a fixed X, while `father(Z,Y)` has many answers for a free Z and Y. This gives us a program which is more efficient for this mode of usage:

```
grandfather(Z,X) :- parent(Y,X), father(Z,Y).
```

When considering the ordering of literals it is important to be aware that reordering can change the mode of usage of the reordered literals, so a call which was deterministic before reordering may become non-deterministic. Because of this, determining the best literal ordering is not always straightforward.

7.5 Adding Redundant Constraints

One useful technique for improving the efficiency of a CLP program is to add constraints which do not change the answers of a program, but which prune unsuccessful branches from the derivation tree earlier.

Definition 7.4
A constraint which can be removed from a rule in a program without changing the answers to any goal is said to be *redundant*.

There are two main types of redundant constraints which can be added to improve the performance of a CLP program.

The first type of redundant constraint is one that is redundant with respect to the answers of a rule's body. We call such constraints *answer redundant*. Given a rule

$$H \text{ :- } L_1, \ldots, L_i, L_{i+1}, \ldots, L_n.$$

it can be useful to add a primitive constraint c to the rule giving

$$H \text{ :- } L_1, \ldots, L_i, c, L_{i+1}, \ldots, L_n.$$

if, for some constraint store C satisfying the intended mode of usage of H, the state $\langle L_1, \ldots, L_i, c \mid C\rangle$ finitely fails while the state $\langle L_1, \ldots, L_i \mid C\rangle$ does not; and every answer to the original body, $L_1, \ldots, L_i, L_{i+1}, \ldots, L_n$, implies that c holds.

The constraint c is redundant since it does not change the answers of the rule for H but, since failure may occur earlier, adding it can make computation more efficient.

The constraint `N ≥ 1` added to the final `sum` program in Section 7.2, is an example of an answer redundant constraint. It is redundant because all answers to `sum(N-1,S)` imply $N \geq 1$ and, so, adding it does not change the answers. Moreover, adding this constraint is useful, since it can cause failure to occur earlier, as witnessed for the goal `sum(1,0)`. Indeed, as we saw, adding this constraint drastically improves the performance of the program by pruning an infinite derivation.

As another example, imagine we now wish to extend the `sum` program to work efficiently for another mode of usage in which we use the program to answer queries of the form, "Is some fixed number S the sum of the first N numbers for some N?" Given the goal `sum(N,6)`, the final `sum` program from Section 7.2 will correctly find that $N = 3$. The goal `sum(N,7)`, however, will not terminate.

Consider the first four states in the simplified derivation for the goal `sum(N, 7)` resulting when the second rule is always selected.

$$\begin{array}{c}
\langle sum(N,7) \mid true\rangle \\
\Downarrow S8 \\
\langle sum(N',S') \mid N = N' + 1 \wedge S' = 6 - N' \wedge N' \geq 0\rangle \\
\Downarrow S8 \\
\langle sum(N'',S'') \mid N = N'' + 2 \wedge S'' = 4 - 2 * N'' \wedge N'' \geq 0\rangle \\
\Downarrow S8 \\
\langle sum(N''',S''') \mid N = N''' + 3 \wedge S''' = 1 - 3 * N''' \wedge N''' \geq 0\rangle \\
\Downarrow S8 \\
\langle sum(N'''',S'''') \mid N = N'''' + 4 \wedge S'''' = -3 - 4 * N'''' \wedge N'''' \geq 0\rangle
\end{array}$$

None of these constraints is ever unsatisfiable and so the derivation is infinite. If we examine the constraint in the last state shown, clearly S'''' can only take negative values. But we know that no sum of numbers from 0 to n can be negative, so we know that this state should fail. The problem is that the program does not contain this extra information we know about the sum: that for every n the sum $0 + 1 + 2 + \cdots + n$ is greater than or equal to zero. We can modify the program to include this information by adding the answer redundant constraint `S ≥ 0`:

```
sum(0,0).                                              (S9)
sum(N, S+N)   :-   N ≥ 1, S ≥ 0, sum(N-1, S).   (S10)
```

With this new program, execution of the goal `sum(N,7)` will now finite fail. The constraint store becomes unsatisfiable at the state corresponding to the last one shown above.

$$\langle sum(N,7) \mid true \rangle$$
$$\Downarrow S10$$
$$\langle sum(N',S') \mid N = N' + 1 \wedge S' = 6 - N' \wedge N' \geq 0 \wedge N' \leq 6 \rangle$$
$$\Downarrow S10$$
$$\langle sum(N'',S'') \mid N = N'' + 2 \wedge S'' = 4 - 2 * N'' \wedge N'' \geq 0 \wedge N'' \leq 2 \rangle$$
$$\Downarrow S10$$
$$\langle sum(N''',S''') \mid N = N''' + 3 \wedge S''' = 1 - 3 * N''' \wedge N''' \geq 0 \wedge N''' \leq 1/3 \rangle$$
$$\Downarrow S10$$
$$\langle \blacksquare \mid false \rangle$$

This program will answer goals in which the second argument S is fixed by constructing a derivation tree which is linear in S. This is because, at each stage in the derivation after the second state, there is an upper bound on the value of the first argument. Each application of the second rule will decrease the upper bound on the first argument until it conflicts with the constraint $N \geq 1$.

The second type of redundant constraints, "solver redundant" constraints, are useful to consider when the underlying solver is incomplete. A constraint is *solver redundant* if it is implied by the current constraint store. It may be useful to add a solver redundant constraint if it makes explicit information which is implied by the store but which the solver, because of incompleteness, is incapable of extracting.

Understanding the incompleteness of the underlying CLP solver is one of the more difficult tasks for the constraint programmer. Unfortunately such understanding is often necessary as incompleteness may mean a constraint does not prune the search as early as intended since the solver will be unable to determine that the constraint store is unsatisfiable. In such circumstances it is often useful to add another constraint which is redundant with regard to the constraints in the store but which causes failure.

Consider the factorial program from Chapter 4:

```
fac(0, 1).                                   (F1)
fac(N, N*F)   :-   N ≥ 1, fac(N - 1, F).   (F2)
```

If we execute the goal `fac(N, 7)`, it will run forever for an analogous reason to the goal `sum(N, 7)` discussed above, namely the current program does not contain the information that all factorials are 1 or greater. Adding the answer redundant constraint $F \geq 1$, we obtain

```
fac(0, 1).                                               (F3)
fac(N, N*F)  :-  N ≥ 1, F ≥ 1, fac(N - 1, F).           (F4)
```

We might now expect that the goal `fac(N, 7)` will fail. Unfortunately, it does not. Let us examine the simplified derivation to see why.

$$\langle fac(N,7) \mid true\rangle$$
$$\Downarrow F4$$
$$\langle fac(N-1,F') \mid F' \geq 1 \land N \geq 1 \land 7 = N \times F'\rangle$$
$$\Downarrow F4$$
$$\langle fac(N-2,F'') \mid F'' \geq 1 \land N \geq 2 \land 7 = N \times (N-1) \times F''\rangle$$
$$\Downarrow F4$$
$$\langle fac(N-3,F''') \mid F''' \geq 1 \land N \geq 3 \land 7 = N \times (N-1) \times (N-2) \times F'''\rangle$$
$$\Downarrow F4$$
$$\langle fac(N-4,F'''') \mid F'''' \geq 1 \land N \geq 4 \land 7 = N \times (N-1) \times (N-2) \times (N-3) \times F''''\rangle$$
$$\Downarrow F4$$
$$\vdots$$

The constraint in the last state shown is unsatisfiable (since $F'''' \geq 1$ and $N \geq 4$ we have that $N \times (N-1) \times (N-2) \times (N-3) \times F''''$ must be greater than 24). Unfortunately, solvers for real arithmetic constraints do not usually handle nonlinear constraints completely, so this unsatisfiability is not detected and the derivation is infinite.

However we can overcome the incompleteness of the constraint solver by adding the constraint that $N \leq F \times N$. This constraint is solver redundant since it is implied by the constraint $N \geq 1 \land F \geq 1$. We must take care when adding this constraint that we take proper account of the way in which the solver handles nonlinear constraints. For instance, the goal `1 = F * N, 2 = F * N` will succeed if different occurrences of the expression $F \times N$ are treated as different expressions by the incomplete solver. This is usually the case. Therefore, rather than use the expression `F * N` twice, we need to name the expression and use the name twice. The new program is:

```
fac(0, 1).                                                        (F5)
fac(N, FN) :- FN = F*N, N ≥ 1, F ≥ 1, N ≤ FN, fac(N - 1, F).  (F6)
```

The new variable FN names the expression $F \times N$. For this program the goal `fac(N, 7)` does fail, after 7 selections of a `fac` literal. The constraints after the 4th selection are still unsatisfiable (although the solver does not detect it). One effect of each derivation step is to increase a lower bound on N by one. From the first derivation step we have that $N \leq FN$ and $FN = 7$. Eventually, the effect of the derivation steps is to constrain $N \geq 8$ and so the derivation fails.

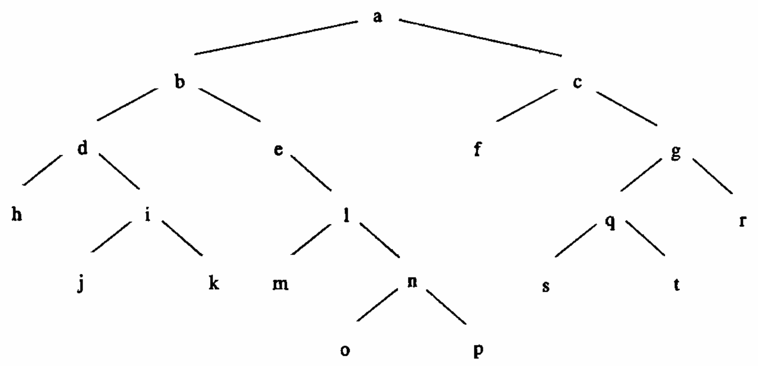


Figure 7.4 A tree for leaf finding.

7.6 Minimization

Much combinatorial search is hidden inside minimization literals. It is, therefore, important to ensure that the goal being minimized has the smallest possible search space.

One common mistake for the novice CLP programmer is to assume that the search space for `minimize(G,E)` for a particular mode of usage is simply that of G for that mode of usage. However, because of the implementation of minimization, the minimization subgoal will be evaluated using goals of the form

```
E < m, G.
```

where m is fixed. Thus it is important to ensure that the definition of G is also efficient for this mode of usage in which E is bounded above.

Consider a simple program which takes a binary tree and returns each leaf of the tree and its corresponding depth.

```
leaflevel(node(null, X, null), X, 0).                      (LL1)
leaflevel(node(TL, _, _), X, D+1) :- leaflevel(TL, X, D).  (LL2)
leaflevel(node(_, _, TR), X, D+1) :- leaflevel(TR, X, D).  (LL3)
```

The first rule succeeds when the tree is a leaf node and the remaining rules return the leaves of the left and right subtrees of a non-leaf node, appropriately increasing the depth. The goal `leaflevel(`t_a`, X, D)` where t_a is the term representation of the tree rooted at a shown in Figure 7.4 has the answers:

$$X = h \land D = 3,\quad X = j \land D = 4,\quad X = k \land D = 4,$$
$$X = m \land D = 4,\quad X = o \land D = 5,\quad X = p \land D = 5,$$
$$X = f \land D = 2,\quad X = s \land D = 4,\quad X = t \land D = 4,$$
$$X = r \land D = 3.$$

Suppose we wish to find the leaves at the minimum depth. The goal

```
minimize(leaflevel(tₐ, X, D), D).
```

will answer this question. Unfortunately, evaluation of this goal will be rather inefficient for most implementations of minimization search.

To understand why, we first need to understand how `minimize` is usually implemented. Most implementations do not traverse the entire derivation tree of the minimization subgoal to find the minimum value of the objective function. Instead, they keep track of the minimum value found so far, and add a constraint to the constraint store which places this value as an upper bound on the objective function. With luck, this upper bound will substantially prune the derivation tree of the original subgoal. This is analogous to the techniques used in retry_int_opt and back_int_opt in Chapter 3 and will be discussed more fully in Section 10.5.

Now we can identify the problem with the current definition of `leaflevel`. Adding a constraint of the form $D < n$ in which n is fixed will not reduce the search space of the minimization subgoal `leaflevel(t, X, D)` since a constraint of this form can only cause failure when a leaf node is reached. Thus, evaluation of

```
minimize(leaflevel(tₐ, X, D), D).
```

will be forced to traverse the entire tree shown in Figure 7.4. This is despite the fact that once a solution has been found in which $D = n$ then the tree does not need to be visited at depths greater than n.

To improve the efficiency of the minimization subgoal we need to modify the predicate `leaflevel` so that it is more efficient for calls in which the second argument is bounded above. Currently such information is not used by the `leaflevel` program to prune the derivation tree. To see exactly why, consider the goal

```
D < 3, leaflevel(tₐ, X, D)
```

and its simplified derivation

$$\begin{array}{c}
\langle D < 3, leaflevel(t_a, X, D) \mid true\rangle \\
\Downarrow \\
\langle leaflevel(t_a, X, D) \mid D < 3\rangle \\
\Downarrow\ LL2 \\
\langle leaflevel(t_b, X, D-1) \mid D < 3\rangle \\
\Downarrow\ LL2 \\
\langle leaflevel(t_d, X, D-2) \mid D < 3\rangle \\
\Downarrow\ LL3 \\
\langle leaflevel(t_i, X, D-3) \mid D < 3\rangle \\
\Downarrow\ LL3 \\
\langle leaflevel(t_k, X, D-4) \mid D < 3\rangle \\
\Downarrow\ LL1 \\
\langle \blacksquare \mid D < 3 \land D - 4 = 0\rangle
\end{array}$$

where t_b, t_d, t_i, t_k are the subtrees of the tree illustrated in Figure 7.4, rooted at b, d, i and k respectively. We would like the derivation to fail before visiting i and k since any leaf in the subtree of t_d must occur at a depth greater than 3.

To do this we can modify the program by adding the answer redundant information that every leaf level is non-negative. Now an upper bound on depth can cause failure before a leaf is reached. The revised program is:

```
leaflevel(node(null,X,null),X,0).                                  (LL4)
leaflevel(node(TL, _, _), X, D+1) :- D ≥ 0, leaflevel(TL, X, D).(LL5)
leaflevel(node(_, _, TR), X, D+1) :- D ≥ 0, leaflevel(TR, X, D).(LL6)
```

The corresponding derivation for the goal `D < 3, leaflevel(`t_a`, X, D)` is

$$\begin{array}{c}
\langle D < 3, leaflevel(t, X, D) \mid true \rangle \\
\Downarrow \\
\langle leaflevel(t_a, X, D) \mid D < 3 \rangle \\
\Downarrow \ LL5 \\
\langle leaflevel(t_b, X, D-1) \mid D < 3 \wedge D \geq 1 \rangle \\
\Downarrow \ LL5 \\
\langle leaflevel(t_d, X, D-2) \mid D < 3 \wedge D \geq 2 \rangle \\
\Downarrow \ LL6 \\
\langle \blacksquare \mid D < 3 \wedge D \geq 3 \rangle
\end{array}$$

As required, the derivation fails before visiting nodes i and k.

Evaluation of the minimization subgoal `minimize(leaflevel(`t_a`, X, D), D)` with this revised program will be substantially faster than with the original program. A typical implementation of `minimize` will visit only half the nodes in the tree t_a since once leaf node h is found, subtrees below depth 3 will not be explored and once leaf node f is found, subtrees below depth 2 will not be explored.

It is not always possible for a minimization search to benefit from upper bounds placed on the objective function because of answers found previously. For example, imagine searching for the deepest leaf in the tree t_a using the goal

```
    minimize(leaflevel(t_a, X, D), -D).
```

Knowing there is a leaf at depth 3 cannot save us from examining every node in the tree to see if it contains a deeper leaf. In other words, there is no more efficient way to write `leaflevel` for the mode of usage in which the depth is bounded from below. For this goal, the original formulation of `leaflevel` is arguably superior since it uses less constraints.

7.7 Identifying Deterministic Subgoals

We have identified a number of guidelines the programmer can follow for writing more efficient CLP programs. However, the default evaluation mechanism still uses depth-first search, which has substantial unnecessary overhead when evaluating deterministic calls. This is because evaluation of even a deterministic call can set up choicepoints to which subsequent execution may backtrack. This is expensive, both in terms of space and time, since the system needs to save information about each choicepoint. In the case of deterministic calls, this cost is unnecessary since, once a deterministic call has succeeded, there is no need to backtrack to a choicepoint within the call as trying other choices can only lead to failure.

We now look at two constructs—*if-then-else* and *once*—provided by many CLP languages which may be used by the programmer to identify pieces of code which are deterministic and so allow the system to evaluate these parts of the program more efficiently.

7.7.1 If-Then-Else

The conditional statement or *if-then-else* is the most common way of identifying deterministic subgoals.

Definition 7.5
An *if-then-else literal* is of the form $(G_{test}$ `->` G_{then} `;` $G_{else})$ where G_{test}, G_{then} and G_{else} are goals. It is read as "*if* G_{test} *then* G_{then} *else* G_{else}."

Evaluation of a conditional literal $(G_{test}$ `->` G_{then} `;` $G_{else})$ is reasonably intuitive. First, the test subgoal G_{test} is evaluated to see if it succeeds. If it does, the *then* branch, G_{then}, is executed. Otherwise, if G_{test} finitely fails, the *else* branch, G_{else}, is executed. We extend the standard CLP evaluation mechanism by adding a derivation step for processing if-then-else literals:

Definition 7.6
An *if-then-else derivation step* is a derivation step $\langle G_1 \mid C_1\rangle \Rightarrow \langle G_2 \mid C_2\rangle$, in which an if-then-else literal is rewritten. Conceptually it is defined as follows (we shall explore the implementation further in Chapter 10). Let G_1 be the sequence of literals

$$L_1, L_2, \ldots, L_m$$

and let L_1 be of the form $(G_{test}$ `->` G_{then} `;` $G_{else})$. There are two cases.

1. The state $\langle G_{test} \mid C_1\rangle$ succeeds and the leftmost successful derivation is

$$\langle G_{test} \mid C_1\rangle \Rightarrow \cdots \Rightarrow \langle \square \mid C\rangle.$$

In this case, C_2 is C and G_2 is $G_{then}, L_2, \ldots, L_m$.

2. Evaluation of the state $\langle G_{test} \mid C_1 \rangle$ finitely fails. In this case, C_2 is C_1 and G_2 is $G_{else}, L_2, \ldots, L_m$.

Note that if evaluation of the test goal G_{test} does not terminate, the original goal $\langle G_1 \mid C_1 \rangle$ cannot be rewritten.

The *if-then-else* literal does not backtrack through different answers for the test goal G_{test}. It simply finds the first answer and commits to this. However, backtracking can occur inside goals G_{then} and G_{else}. The main benefit of *if-then-else* literals is that once the *then* branch is chosen, there is no need to backtrack to the *else* branch. Thus, there is no need for the system to set up a choicepoint.

As an example, let us define a predicate `abs(X,Y)` constraining Y to be the absolute value of X.

```
abs(X, Y) :- (X ≥ 0 -> Y = X ; Y = -X).
```

This has the natural reading: if $X \geq 0$ then the absolute value of X is X, else the absolute value of X is $-X$.

As an example of its evaluation, the goal `abs(4, A)` succeeds with the single answer $A = 4$ as expected. Execution starts by rewriting `abs(4,A)` to the state

$$\langle 4 = X, A = Y, (X \geq 0 \texttt{ -> } Y = X \texttt{ ; } Y = -X) \mid \mathit{true} \rangle.$$

The constraints $4 = X$ and $A = Y$ are added in turn to the constraint store giving

$$\langle (X \geq 0 \texttt{ -> } Y = X \texttt{ ; } Y = -X) \mid 4 = X \land A = Y \rangle.$$

The if-then-else literal is processed by first evaluating the state

$$\langle X \geq 0 \mid 4 = X \land A = Y \rangle.$$

This has a single successful derivation whose final constraint store is

$$4 = X \land A = Y \land X \geq 0.$$

Thus the state above is rewritten using the *then* branch of the if-then-else literal giving:

$$\langle Y = X \mid 4 = X \land A = Y \land X \geq 0 \rangle.$$

Finally, the constraint $Y = X$ is added to the constraint store, giving the answer.

Similar reasoning shows that the goal `abs(-4, A)` succeeds with answer $A = 4$ as expected.

To understand the advantage of if-then-else, consider how we would have had to write the `abs` program without it. This would have required two rules:

```
abs(X, X) :- X ≥ 0.
abs(X, -X) :- X < 0.
```

Evaluation of the goal abs(4, A) using this program will also succeed with the answer $A = 4$ as expected. However, unlike the definition using if-then-else, there will also be a choicepoint set up which, on subsequent failure, will cause the second rule in this definition of abs to be tried. This will of course fail. The advantage of the if-then-else definition is that is tells the CLP system that the definition for abs is deterministic in the sense that, if the test succeeds, there is no need to try the *then* branch since this will fail (once the implicit negation of the test is added).

Unfortunately, the programmer must be very careful when using if-then-else literals as their behaviour depends greatly on their mode of usage. One problem is that if the arguments to the test goal in an if-then-else literal are not fixed at the time of evaluation, answers may be lost.

The definition of abs using if-then-else works well for the mode of usage in which the first argument, X, is fixed. However, if X is not fixed then its behaviour is quite strange. For instance, the goal

```
    abs(X, 2), X < 0.
```

fails, rather than giving the expected answer $X = -2$. This is because the test goal $X \geq 0$ succeeds, which commits the derivation to use the *then* part of the if-then-else subgoal $Y = X$. Later when we add $X < 0$ the derivation fails. This means that the order of the literals becomes important in reasoning with if-then-else literals since, for example, the goal

```
    X < 0, abs(X, 2).
```

succeeds with answer $X = -2$.

Another potential problem with if-then-else is that only the *first* answer of the state $\langle G_{test} \mid C_1 \rangle$ is found and afterwards no other answers will be examined.

Suppose we wish to define a constraint between X and Y that holds if they are at least 4 apart or they are identical. We might use if-then-else to write this as:

```
far_or_equal(X, Y) :- (apart(X,Y,4) -> true ; X = Y).

apart(X,Y,D) :- X ≥ Y + D.
apart(X,Y,D) :- Y ≥ X + D.
```

Here, for the first time in a program, we use the primitive constraint true. Normally a *true* constraint adds nothing to the goal, so it is not used. But here we need to provide a goal G_{then} which does nothing. Therefore we use true. When it is processed it is simply removed from the goal.

The program states that either X and Y are at least 4 apart, or if this is not the case they are equal. The goal far_or_equal(1, 6) succeeds, and the goal far_or_equal(1, 3) fails as expected. They work correctly because the mode of usage ensures all variables in the test goal are fixed. The goal

```
    far_or_equal(1, Y), Y = 6.
```

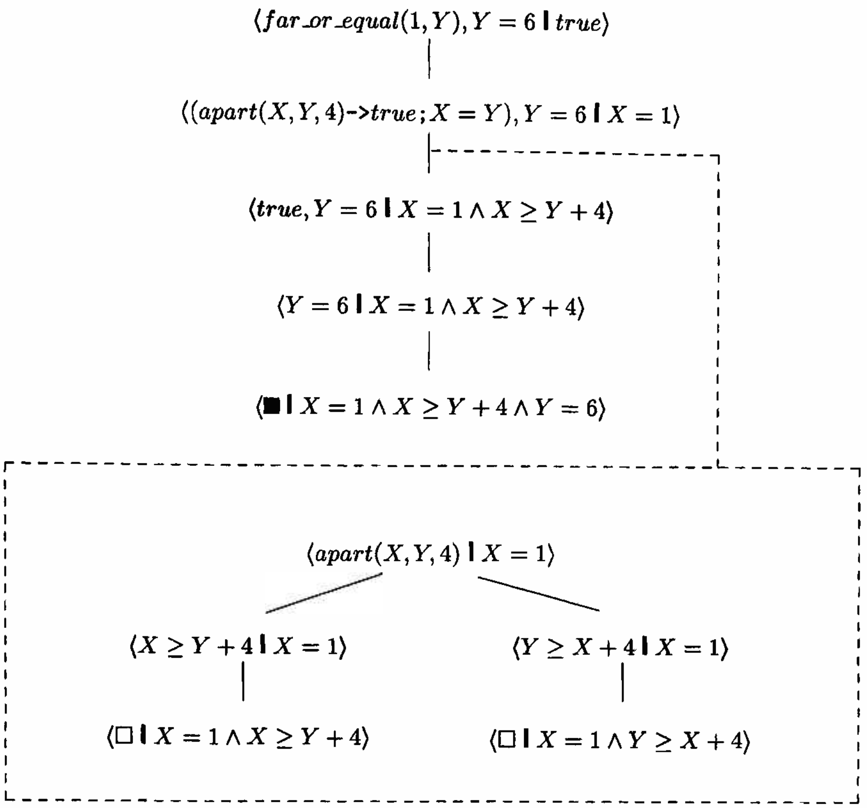

$\langle far\_or\_equal(1,Y), Y = 6 \mid true\rangle$
$\langle (apart(X,Y,4)\text{->}true; X = Y), Y = 6 \mid X = 1\rangle$
$\langle true, Y = 6 \mid X = 1 \land X \geq Y + 4\rangle$
$\langle Y = 6 \mid X = 1 \land X \geq Y + 4\rangle$
$\langle \blacksquare \mid X = 1 \land X \geq Y + 4 \land Y = 6\rangle$
$\langle apart(X,Y,4) \mid X = 1\rangle$
$\langle X \geq Y + 4 \mid X = 1\rangle$
$\langle Y \geq X + 4 \mid X = 1\rangle$
$\langle \square \mid X = 1 \land X \geq Y + 4\rangle$
$\langle \square \mid X = 1 \land Y \geq X + 4\rangle$

Figure 7.5 Simplified derivation tree for `far_or_equal(1, Y), Y = 6.`

fails however. The derivation tree shown in Figure 7.5 illustrates why. At the state in which the if-then-else literal is rewritten, the system first explores the derivation tree for the goal $\langle apart(X,Y,4) \mid X = 1\rangle$. This tree is shown in the dashed box. Execution of the test goal `apart(X,Y,4)` traverses this derivation tree until the first answer, $X = 1 \land X \geq Y + 4$, is found. This gives the constraint store for the subsequent execution of the then goal. But this answer is not compatible with the constraint $Y = 6$, so the derivation eventually fails. However, if the other answer to the test goal had been used the derivation would have succeeded.

Although if-then-else is dangerous to use when the mode of usage does not ensure all the variables involved in the test are fixed, it can be very useful if all the variables are fixed. In particular, once we test a constraint in which all of the variables are fixed and it fails then we can be sure that the negation of the constraint holds, so we do not need to test for this explicitly.

For example, we can redefine the `cumul_predecessors` predicate of Example 6.5 to avoid using `not_member` since its only purpose was to ensure that the `member` constraint did not hold. The new code is:

```
cumul_predecessors_ift([], _, Pre, Pre).
cumul_predecessors_ift([N|Ns], AList, Pre0, Pre) :-
    (member(N, Pre0) ->
        Pre1 = Pre0
    ;
        predecessors(N, AList, [N|Pre0], Pre1)
    ),
    cumul_predecessors_ift(Ns, AList, Pre1, Pre).
```

If N has already been seen, that is, it is a member of $Pre0$, then the predecessors do not change so $Pre1 = Pre0$. Otherwise N is added to the accumulator and its predecessors are recursively added to obtain $Pre1$. Computation continues on the rest of the list Ns with the current accumulator $Pre1$. Correctness of this definition relies on N and $Pre0$ being fixed by the time `member(N, Pre0)` is evaluated.

7.7.2 Once

A construct closely related to if-then-else is `once`. This finds the first answer to a subgoal and commits to it.

Definition 7.7
A *once literal* is of the form $once(G)$ where G is a goal, called the *once subgoal.*

The operational behavior of a once literal is quite straightforward: find the first answer to the once subgoal. If the once subgoal finitely fails then fail.

Definition 7.8
A *once derivation step* is a derivation step $\langle G_1 \mid C_1 \rangle \Rightarrow \langle G_2 \mid C_2 \rangle$ in which a once subgoal is rewritten. Conceptually it is defined as follows (again we explore the implementation further in Chapter 10). Let G_1 be the sequence of literals

$$L_1, L_2, \ldots, L_m$$

and let L_1 be of the form $once(G)$. There are two cases.

1. The state $\langle G \mid C_1 \rangle$ succeeds and has the leftmost successful derivation

$$\langle G \mid C_1 \rangle \Rightarrow \cdots \Rightarrow \langle \Box \mid C \rangle.$$

In this case, C_2 is C and G_2 is $L_2, \ldots, L_m$.

2. The state $\langle G \mid C_1 \rangle$ finitely fails. In this case, C_2 is $false$ and G_2 is $\blacksquare$.

The `once` operator is used when the programmer knows that each of the possible answers to some user-defined constraint are equivalent. Hence, once the first answer is found the other answers can be ignored.

Example 7.1

Consider the following definition for a predicate `intersect` which determines if two sets, which are represented by lists, intersect.

```
intersect(L1, L2) :- member(X,L1), member(X,L2).
```

Now consider the goal

```
    intersect([a,b,e,g,h], [b,e,f,g,i]).
```

Part of the simplified derivation tree for this goal is shown in Figure 7.6. The total simplified derivation tree has 72 states. The first answer, *true*, is found after visiting 18 states. Exploration of the remaining part of the tree finds 2 more answers, both *true*.

Since all of the answers are equivalent, this definition of `intersect` can be improved by using a once literal.

```
intersect(L1, L2) :- once(member(X,L1), member(X,L2)).
```

The advantage of this formulation is that it tells the CLP system not to bother looking for alternate answers after finding the first answer. This can dramatically reduce the search space.

For example, using the original formulation of `intersect`, evaluation of the goal

```
    intersect([a,b,e,g,h], [b,e,f,g,i]), 0 = 1.
```

searches a simplified derivation tree analogous to that partially illustrated in Figure 7.6, visiting a total of 72 states, before finite failing. With the improved formulation, only those (18) states in the derivation tree before the first answer are visited.

7.8 An Extended Example: Bridge Building

We conclude this chapter with an extended example that highlights many of the efficiency issues we have discussed. One of the strengths of CLP languages is in engineering analysis and design. We have already seen how arithmetic constraints and hierarchical definitions allow the natural and powerful modelling of complex systems. Design is facilitated by the built-in capacity for search. By first building a general program for analysis and then adding a component to search through designs, we can write simple yet powerful design programs. However, for this to be practical it is important that the design space is as small as possible. In this section we show how techniques for reducing the search space of a program can be used to develop a program for designing "spaghetti bridges."

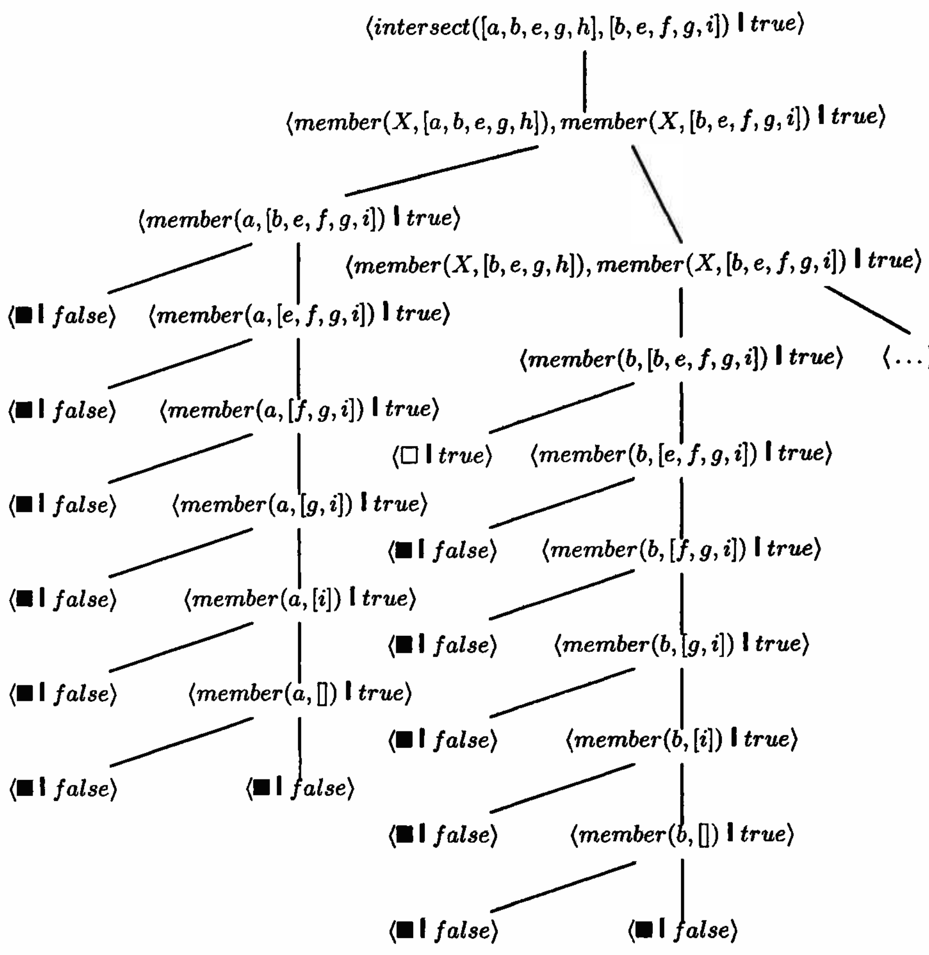

⟨intersect([a, b, e, g, h], [b, e, f, g, i]) | true⟩
⟨member(X, [a, b, e, g, h]), member(X, [b, e, f, g, i]) | true⟩
⟨member(a, [b, e, f, g, i]) | true⟩
⟨member(X, [b, e, g, h]), member(X, [b, e, f, g, i]) | true⟩
⟨■ | false⟩
⟨member(a, [e, f, g, i]) | true⟩
⟨member(b, [b, e, f, g, i]) | true⟩
⟨...⟩
⟨■ | false⟩
⟨member(a, [f, g, i]) | true⟩
⟨□ | true⟩
⟨member(b, [e, f, g, i]) | true⟩
⟨■ | false⟩
⟨member(a, [g, i]) | true⟩
⟨■ | false⟩
⟨member(b, [f, g, i]) | true⟩
⟨■ | false⟩
⟨member(a, [i]) | true⟩
⟨■ | false⟩
⟨member(b, [g, i]) | true⟩
⟨■ | false⟩
⟨member(a, []) | true⟩
⟨■ | false⟩
⟨member(b, [i]) | true⟩
⟨■ | false⟩
⟨■ | false⟩
⟨■ | false⟩
⟨member(b, []) | true⟩
⟨■ | false⟩
⟨■ | false⟩

Figure 7.6 Simplified derivation tree for an intersection goal.

7.8.1 Analysis

Consider the problem of analysing a bridge made out of spaghetti. For simplicity we shall reduce the problem to two dimensions by ignoring the width of the bridge. The bridge is built out of spaghetti struts and joins, which have the following properties.

A strut can be stressed with an amount F of stretching force, parallel in direction to its length. A strut of length L cm can sustain any amount of stretching force F, but only $0.5 \times (6 - L)$ Newton (N) of compression force, that is negative stretching force. Implicitly this constrains each strut to be no longer than 6 cm. The total amount of spaghetti available to build struts is 20 cm.

Two fixed joins are available on each side of the gap to be spanned by the bridge. Other joins are floating joins, one of which must be placed in the center of the gap. For stability every floating join must be connected to at least three struts of

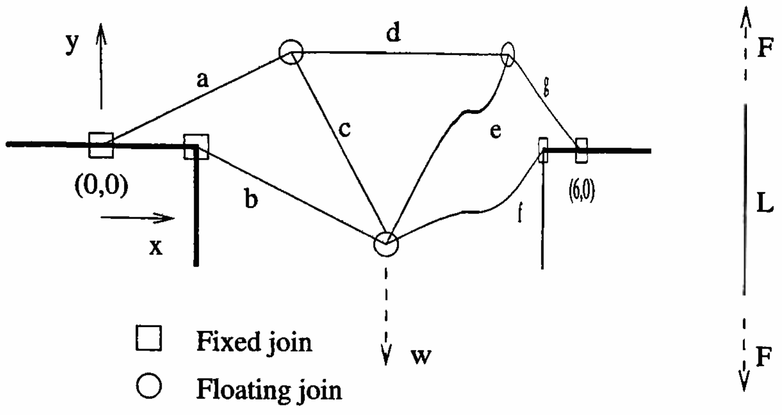


Figure 7.7 Building a spaghetti bridge.

spaghetti. An exception is the center join which only needs to be connected to two struts, since the weight hanging from the bridge acts as a third strut. A floating join can sustain any amount of compression force from any one strut, but only 2 N of stretching force. The vector sum of the forces at each floating join must equal zero. The bridge is adequate if it can support a weight of 2 N hanging from the center floating join. A sample bridge is shown in Figure 7.7.

To solve this problem we first need to decide how to represent a bridge. Let us use two lists, a list of floating joins and a list of struts which act as association lists detailing the data associated with struts and joins in a bridge. The centre join is represented by the term *cjoin*(x, y, l) where (x, y) is the coordinates of the join and l is a list of the names of connected struts. Other floating joins are represented by *join*(x, y, l). This representation allows us to test whether a join J is the centre join by simply using the constraint $J =$ *cjoin*$(\_,\_,\_)$. A strut is represented by the term *strut*(n, x_1, y_1, x_2, y_2) where n is the strut name, and the two endpoints are (x_1, y_1) and (x_2, y_2). The bridge shown in Figure 7.7 is represented by the list of joins, abbreviated to *js*,

```
[join(2,1,[a,c,d]), join(4,1,[d,e,g]), cjoin(3,-1,[b,c,e,f])]
```

and the list of struts, abbreviated to *ss*,

```
[    strut(a,0,0,2,1), strut(b,1,0,3,-1), strut(c,2,1,3,-1),
     strut(d,2,1,4,1), strut(e,3,-1,4,1), strut(f,3,-1,5,0),
     strut(g,4,1,6,0)]
```

Analysis of a bridge produces an association list detailing the forces on each strut. Records in the list are pairs of form $f(N, F)$ where N is the strut name and F the stretching force on the strut. A program which analyses a bridge design must simply collect the constraints on forces acting on each join and strut.

We begin by analyzing the forces in each strut. For each strut we must assert the maximum compression force constraint, and for the total group of struts we need to assert the total length constraint. The following program defines a predicate struts_cons which takes the list of struts and asserts the maximum compression constraints and calculates the total length of the struts.

```
struts_cons([], [], 0).
struts_cons([strut(N,X1,Y1,X2,Y2) | Ss], [f(N, F) | Fs], TL) :-
    L = √(X1-X2)*(X1-X2) + (Y1-Y2)*(Y1-Y2),
    F ≥ - 0.5 * (6 - L),
    TL = L + RL,
    struts_cons(Ss, Fs, RL).
```

This is a simple recursive program that adds constraints for each strut. The first argument is a list of strut information, as in the example above. The second argument is an association list relating the strut name to the force within the strut. The third argument is the total length of the struts in the list. The non-recursive rule is first, the usual rule ordering, so that the program is likely to work in more modes of usage (for example, the goal struts_cons(Ss, Fs, TL) will find answers before reaching an infinite derivation). The constraints in the recursive case are all placed before the recursive call in case they do not hold. The program is deterministic as long as either the first or second argument is a list of fixed length.

The goal struts_cons(*ss*, Fs, TL) succeeds with answer

$$\begin{array}{rcl} Fs & = & [f(a,F_a),f(b,F_b),f(c,F_c),f(d,F_d),f(e,F_e),f(f,F_f),f(g,F_g)] \ \land \\ TL & = & 15.4164 \ \land \\ \multicolumn{3}{l}{F_a \geq -1.88197 \ \land \ F_b \geq -1.88197 \ \land \ F_c \geq -1.88197 \ \land \ F_d \geq -2 \ \land} \\ \multicolumn{3}{l}{F_e \geq -1.88197 \ \land \ F_f \geq -1.88197 \ \land \ F_g \geq -1.88197} \end{array}$$

This shows the association list produced and the maximum compression constraints applied to each force associated to a strut, as well as the total length computation.

However this definition is not ideal for our purposes. This predicate will be required to evaluate partially specified designs, and, so as to ensure the design space is a small as possible, should fail as quickly as possible. The current definition does not always do this. Suppose the total length of the struts was limited to 5 cm, then the goal

```
    TL ≤ 5, struts_cons(ss, Fs, TL).
```

will fail. Unfortunately, the derivation will fail only after examining all of the struts. This is the same problem as occurred for the goal sum(N,7) in Section 7.5, although no infinite derivation is possible here.

We know that each strut is of length at least 0, so the total length of any group of struts is also at least 0. Any answer to strut_cons will reflect this, but the information is not available until the last strut is examined. By adding the answer redundant constraint that the remaining length *RL* must be at least 0, the derivation will fail earlier.

```
struts_cons([], [], 0).
struts_cons([strut(N,X1,Y1,X2,Y2) | Ss], [f(N, F) | Fs], TL) :-
    L = √(X1-X2)*(X1-X2) + (Y1-Y2)*(Y1-Y2),
    F ≥ - 0.5 * (6 - L),
    TL = L + RL,
    RL ≥ 0,
    struts_cons(Ss, Fs, RL).
```

The goal `TL ≤ 5, strut_cons(`*ss*`, Fs, TL)` now fails after examining the first three struts.

Now suppose we use the `strut_cons` program to analyse a partial design in which not all coordinates are known. Unfortunately, the constraint solver is incomplete for nonlinear constraints so it may not fail when we expect it to. For example, the goal

```
    Fa ≤ -4, struts_cons([strut(a,A,B,C,D)], [f(a,Fa)], TL).
```

asserts that the compression force in strut a is at least 4 N. Since the maximum compression force sustainable by any strut is 3 N, this goal should fail. However, it succeeds. This is because the incomplete solver does not determine that the length of the strut is at least 0. Adding the solver redundant constraint $L \geq 0$ will solve this problem. In fact we should probably add $L > 0$, which asserts that the length of every strut is greater than 0, since this is an implicit constraint on the design. The final code for `strut_cons` is therefore

```
struts_cons([], [], 0).
struts_cons([strut(N,X1,Y1,X2,Y2) | Ss], [f(N, F) | Fs], TL) :-
    L = √(X1-X2)*(X1-X2) + (Y1-Y2)*(Y1-Y2),
    L > 0,
    F ≥ - 0.5 * (6 - L),
    TL = L + RL,
    RL ≥ 0,
    struts_cons(Ss, Fs, RL).
```

With this definition, the goal above fails as desired.

Now let us examine how to model the constraints on a single join. We need to check that the join has enough incident struts, that the sum of the forces at the join equals zero and that no strut compresses the join more than 2 N. First, let us write a predicate to determine the sum of the forces in the x and y direction at a join.

```
sum_forces([], _, _, _, _, 0, 0).
sum_forces([N|Ns], X, Y, Ss, Fs, SFX, SFY) :-
    member(strut(N, X1, Y1, X2, Y2), Ss),
    whichend(X1,Y1,X2,Y2,X,Y,X0,Y0),
    member(f(N, F), Fs),
    F ≤ 2,
    L = √(X-X0)*(X-X0) + (Y-Y0)*(Y-Y0),
    FX = F * (X - X0) / L,
    SFX = FX + RFX,
    FY = F * (Y - Y0) / L,
    SFY = FY + RFY,
    sum_forces(Ns, X, Y, Ss, Fs, RFX, RFY).

whichend(X, Y, X0, Y0, X, Y, X0, Y0).
whichend(X0, Y0, X, Y, X, Y, X0, Y0).
```

The predicate `sum_forces(Ns, X, Y, Ss, Fs, SFX, SFY)` takes a list of strut names, Ns, that are incident on the join at (X,Y), the list of struts, Ss, and the force association list, Fs. It determines the sum of forces in the x direction, SFX, and in the y direction, SFY. The base case is clear—the sum of the forces in both directions is 0 for an empty list of struts. The recursive case first looks up the information about the strut named N in the strut list. A call to `whichend` determines which end of the strut $s(N, X1, Y1, X2, Y2)$ is connected to the join (X,Y) (where $(X0, Y0)$ is the coordinate of the other end). Next, the force information for the strut is looked up in the force list. A constraint enforcing the stretching force to be no greater than 2 N is added (this implies the compression force on the join is no greater than 2 N). Finally the proportion of the force in the strut exerted in the x and y direction is calculated.

We can use the `sum_forces` predicate to build the constraints on each join. There are two cases to consider. For the center join, the sum of forces must equal the weight, W, in the y direction, and 0 in the x direction. For all other floating joins, both sums must be zero. The following program places these constraints on the joins.

```
joins_cons([], _Ss, _Fs, _W).
joins_cons([J|Js], Ss, Fs, W) :-
    one_join(J, Ss, Fs, W),
    joins_cons(Js, Ss, Fs, W).

one_join(cjoin(X, Y, Ns), Ss, Fs, W) :-
    Ns = [_,_|_],
    sum_forces(Ns, X, Y, Ss, Fs, 0, W).
one_join(join(X, Y, Ns), Ss, Fs, _W) :-
    Ns = [_,_,_| _],
    sum_forces(Ns, X, Y, Ss, Fs, 0, 0).
```

The predicate `joins_cons` checks each join J in the first argument using the predicate `one_join`. The first rule for `one_join` ensures that a center join has at least 2 incident struts by testing that the list Ns has at least two elements, and that the sum of the forces of struts incident at it adds up to W. The second rule ensures other floating joins have at least 3 incident struts (by testing that the list Ns has at least three elements) and the sum of forces incident at the join is zero.

The following goal tests whether the bridge design illustrated in Figure 7.7 satisfies the design constraints.

```
TL ≤ 20, W ≥ 2, struts_cons(ss, Fs, TL), joins_cons(js, ss, Fs, W).
```

The answer restricted to W is $2 \leq W \land W \leq 3.28328$ indicating the bridge satisfies the design constraints.

7.8.2 Design

Since we have written the analysis program with more complex modes of usage in mind, we can also use it for designing bridges. To avoid an explosion in the design search space, we need to add the analysis constraints *before* enumerating different designs, thus constraining the problem before generating possible solutions. This reduces the search space by ensuring that partial designs which do not satisfy the design constraints will be rejected as soon as possible.

Our aim is to design a bridge over a gap of width 4 cm, as shown in Figure 7.7, which can support the maximum possible weight hanging from the centre. However, simply running the goal

```
TL ≤ 20, minimize((struts_cons(Ss,Fs,TL),joins_cons(Js,Ss,Fs,W)),-W).
```

will, unfortunately, not give an answer. There are three problems. First, not all implicit constraints about correct designs are modelled in `strut_cons` and `joins_cons`. For example, each strut must have both endpoints at either a fixed or floating join. Second, even if we added these implicit constraints, the evaluation is unlikely to find any correct designs before it finds an infinite derivation. Third, even if we found a suitable design, the constraints remaining are likely to be nonlinear (because of length calculations) and so it is unlikely that the minimization algorithm will be able to find an optimal solution. What can we do?

The `strut_cons` predicate is deterministic as long as the number of struts is fixed. The `joins_cons` predicate has one solution as long as the join list is fixed in length and a fixed list of distinct incident strut names is given for each join. It is not deterministic because of the search that occurs in the `member` and `whichend` literals. However, by using a once literal we can make it deterministic.

This suggests that we write a user-defined constraint to define suitable bridge topologies. This will fix the number of struts and joins, and ensure that implicit deign constraints, such as struts having endpoints at joins, are satisfied. Then we can apply the strut constraints and join constraints deterministically. Finally we can search for appropriate positions for the joins. The top-level goal will be of form

```
    tpl(Js, Ss, Vs),
    TL ≤ 20,
    struts_cons(Ss,Fs,TL),
    once(joins_cons(Js,Ss,Fs,W)),
    minimize(position(Vs),-W).
```

The predicate `tpl(Js, Ss, Vs)` generates a topology for the bridge. *Js* is a list of the joins with variables for their coordinates, *Ss* is a list of the struts and their connections and *Vs* is the list of variables in the coordinates of the joins. To make it deterministic, the call to `joins_cons` is surrounded with `once`. The predicate `position(Vs)` searches through positions for the coordinate variables.

Unfortunately, automatic generation of meaningful bridge topologies is quite difficult. Even if we only consider the possible graphs for three floating joins there are 2^{15} possibilities. So if we are going to search through different topologies for our bridge we need to restrict this large search space very significantly. One reasonable solution to this problem is to involve the designer in this phase of design. Instead of generating different topologies automatically, a topology can be given by the user. In this way the design program is used interactively, with the user specifying the design topology, and the CLP system finding a good set of coordinates for a bridge that satisfies the given topology.

We shall now continue developing our bridge design program under the assumption that the designer has suggested the bridge has the same topology as the bridge in Figure 7.7. That is, the designer is interested in bridges with the same struts and connection. Since only the topology is decided, the position of the floating joins is not fixed. This can be achieved by defining the predicate `tpl` to be:

```
tpl([join(X1,Y1,[a,c,d]),join(X2,Y2,[d,e,g]),cjoin(X3,Y3,[b,c,e,f])],
    [strut(a,0,0,X1,Y1), strut(b,1,0,X3,Y3), strut(c,X1,Y1,X3,Y3),
    strut(d,X1,Y1,X2,Y2), strut(e,X3,Y3,X2,Y2), strut(f,X3,Y3,5,0),
    strut(g,X2,Y2,6,0)], [X1,Y1,X2,Y2,X3,Y3]) :- X3 = 3.
```

The call to `tpl(Js, Ss, Vs)` constrains the join list, *Js*, strut list, *Ss*, and list of coordinates, *Vs*, (which we shall make use of later). The coordinates of the floating joins are replaced by variables in both the join and strut list. The variables are collected in a list as the third argument. Note that, since we know that the centre join must be in the centre of the gap, we can set $X3 = 3$.

Assuming the constraint solver handles nonlinear constraints incompletely, the goal

```
tpl(Js,Ss,Vs), TL≤20, struts_cons(Ss,Fs,TL), joins_cons(Js,Ss,Fs,W).
```

will, unfortunately, fail to provide useful information since the constraints it sets up are nonlinear because of the length calculations. The answer to the goal is a large nonlinear constraint which is not particularly helpful in finding an actual solution.

We need to design a predicate `position` which will search for coordinate values. Once the coordinates are fixed, the nonlinear constraints become linear and therefore amenable to minimization. One simple restriction which greatly simplifies the search for coordinate values is to restrict the placement of floating joins to integral points, that is integers, in the range [−2..6]. This restriction makes the search space finite. It is important that we perform this non-deterministic search for coordinates *after* the analysis constraints have been added. In this way coordinates for the floating joins that do not satisfy the constraints are eliminated from consideration as soon as possible.

We can build a program to search through the designs as follows:

```
position([]).
position([V|Vs]) :-
    member(V, [6,5,4,3,2,1,0,-1,-2]),
    position(Vs).

bridge_constraints(W, Vs) :-
    tpl(Js, Ss, Vs),
    TL ≤ 20,
    struts_cons(Ss, Fs, TL),
    once(joins_cons(Js, Ss, Fs, W)).

design_goal(W, Vs) :-
    bridge_constraints(W, Vs),
    once(minimize(position(Vs), -W)).
```

The `position` predicate enumerates the 9 possible coordinate values for each variable in its list argument. (We will see more examples of this kind of "labelling" predicate in Section 8.3). The `bridge_constraints` predicate sets up the topology using `tpl`. It constrains the total length of struts to be less than 20 and adds the strut and join constraints. The `design_goal` predicate sets up the constraints and performs a minimization search over the different possible coordinate values. Since only the first answer is required, we use once.

The goal `design_goal(W, Vs)` results in the answer

```
W = 6.15663, Vs = [2, 2, 5, 1, 3, 3].
```

showing that one of the strongest bridges based on the given design topology has joins at $(2, 2)$, $(5, 1)$ and $(3, 3)$.

Unfortunately, the above program is unable to make use of lower bounds on the weight to restrict the search inside the minimize literal, so evaluation of the minimization literal completely traverses the derivation tree of the minimization subgoal. However, since the weight that can be carried at the centre join is intimately connected to forces which rely on the nonlinear length calculations, it seems intrinsic to the problem that the entire design must be determined before an upper bound on the weight can be determined.

We can, however, take advantage of the restricted range of possible coordinate values, to add more redundant constraints. Since the joins are all positioned at integer coordinates, the length of each strut must be at least 1. By adding the answer redundant constraint $L \geq 1$ to `struts_cons` we get more information about the forces possible in each strut and so reduce the search space. We can also add constraints which linearly approximate the length calculation and so reduce the search space. The solver redundant linear constraint

$$X1 - X2 \leq L \land X2 - X1 \leq L \land Y1 - Y2 \leq L \land Y2 - Y1 \leq L,$$

which constrains the length of a strut to be longer than length spanned in both the horizontal and vertical directions, can be added to the definitions of `strut_cons` and `sum_forces`. Because it is linear the solver can use it to deduce information about strut lengths and so reduce the search space.

Restricting the solution to integral coordinates is rather severe. One way of finding better designs for the bridge is to perform a "local search" from the best integral solution. This search examines other coordinate values in the neighbourhood of the current best solution, in order to find a better solution. The search can continue until the new solution found is only slightly better than the previous one, in which case the search can halt. We can think of this local search as "perturbing" the coordinates of the current solution.

Given a best integral solution, we first search for the best bridge whose coordinates vary by 0, +0.5 or −0.5 from their value in the integral solution. We can then vary the coordinates in this solution by 0, +0.25 or −0.25, and so on, until the best perturbed design does not improve very much on the previous one. The following program does this.

```
perturbation([], [], _).
perturbation([V|Vs], [Val|Vals], D) :-
     perturb(V, Val, D),
     perturbation(Vs, Vals, D).

perturb(V, Val, D) :- V = Val - D.
perturb(V, Val, _D) :- V = Val.
perturb(V, Val, D) :- V = Val + D.

pert_goal(FW, FVs) :-
     design_goal(W,Vs),
     improve(Vs, W, FVs, FW, 0.5).

improve(Vs, W, FVs, FW, D) :-
     bridge_constraints(NW, NVs),
     NW ≥ W,
     once(minimize(perturbation(Vs, NVs, D), -NW)),
     ( NW < 1.01 * W ->
          FVs = NVs, FW = NW
     ;
          improve(NVs, NW, FVs, FW, D/2) ).
```

The program works as follows. The predicate `perturbation(Vs, Vals, D)` sets each variable in the list of variables Vs to be equal to its corresponding value in the list $Vals$, or D greater or less. This is the local search. The goal `pert_goal` finds an integral solution using `design_goal` and then uses `improve` with an initial perturbation length of 0.5 to search for a better solution. The predicate `improve(Vs, W, FVs, FW, D)` starts from an initial solution Vs which carries weight W, and finds a final solution FVs which carries weight FW starting with an initial perturbation D. After setting up a new copy of the bridge constraints on NVs and NW, and adding the constraint that the perturbed design must carry at least the same weight as the previous solution, the perturbation search is performed to find a new design. The resulting maximum weight NW is compared with the maximum weight of the previous design. If it fails to improve the previous design by at least 1%, the last design is returned. Otherwise another perturbation step is performed, with a halved perturbation length.

The goal `pert_goal(FW, FVs)` initially discovers a solution

$$W = 6.15663 \wedge Vs = [2, 2, 5, 1, 3, 3].$$

Perturbing this solution by 0.5 finds the solution

$$W = 6.3014 \wedge Vs = [2.5, 2.5, 4.5, 1.5, 3, 3.5]$$

Perturbing this solution by 0.25 finds the same solution. This is returned since it is less than 1% better than the previous solution. The perturbation search is essentially an implementation of a hill climbing approach where the step lengths are decreasing. Note that the search only finds a locally best design which may be far away from the best design.

The above designs are not symmetric. A reasonable requirement of the bridge design is that it be symmetric. That is to say, the two non center floating joins should be positioned at the same height and be horizontally symmetric with respect to the gap. By adding the constraints $Y1 = Y2$ and $X2 = 6 - X1$ to the rule for `tpl` we will only consider symmetric bridges. This reduces the search space considerably and actually leads to the program finding a better final solution.

Using the symmetry constraints the best integral solution found is

$$W = 6.15663 \wedge Vs = [2, 2, 4, 2, 3, 3].$$

Perturbing this by 0.5 finds the solution

$$W = 6.3014 \wedge Vs = [2.5, 2.5, 3.5, 2.5, 3, 3.5].$$

Perturbing this by 0.25 finds the solution

$$W = 6.62524 \wedge Vs = [2.25, 2.75, 3.75, 2.75, 3, 3.75],$$

and perturbing this by 0.125 finds the solution

$$W = 6.63839 \land Vs = [2.125, 2.625, 3.875, 2.625, 3, 3.75].$$

The last solution is less than 1% better than the previous solution so it is returned. The best bridge we have discovered therefore has joins at (2.125, 2.625), (3.875, 2.625) and (3, 3.75). It improves on the best integral solution by approximately 7.5%.

7.9 Summary

In this chapter we have studied how constraint programmers can understand the efficiency of their programs and have described various techniques which can be used to improve efficiency.

Broadly speaking, efficiency is measured in terms of the size of a goal's derivation tree. Since the size of the derivation tree depends greatly on the type of goal being solved, efficiency depends on the intended mode of usage of a predicate. Understanding the shape of the derivation tree and the way in which it is explored is particularly important when writing recursive programs. By understanding the shape of the derivation tree for a given mode of usage, the programmer can ensure that a solution will be found before an infinite branch is reached. To fully understand the shape of the derivation tree, the programmer also needs to know for which types of constraints the underlying solver is incomplete.

Although the answers for the goal are independent of the order in which literals appear in goals, the size and shape of the derivation tree greatly depends on the order of rules in the program and order of literals in the rule bodies. We have examined how to order rules and literals in rule bodies so as to make programs more efficient. One guideline is that non-recursive rules should appear before recursive rules in a predicate definition. When ordering literals in a rule body, it is better to place constraints at the earliest point at which they may cause failure and to place user-defined constraints with fewer answers, in particular deterministic calls, earlier than those with many answers.

Another strategy we have investigated for improving efficiency is to add redundant constraints. These come in two flavours. Constraints which are redundant with respect to the "future" answers and constraints which are redundant with respect to the current constraint store. In the next chapter we shall see that adding redundant constraints is an important technique for improving the efficiency of CLP programs computing over finite domains.

We have also looked at two constructs—if-then-else and once—provided by many CLP systems to identify sub-computations which do not require backtracking. The programmer must be careful when using if-then-else as it is dangerous if the mode of usage does not ensure the test has, at most, one answer and the variables are fixed. However, if these conditions are met, it can be quite useful.

7.10 Exercises

7.1. What is the size of the derivation tree of the factorial program from Chapter 4 for goals in which the first argument to `fac` is fixed?

7.2. Give the simplified derivation trees for `grandfather(X,peter)` using the two different programs defining `grandfather`.

7.3. Consider the following program defining predicate `anc(Y,X)` which holds if Y is an ancestor of X.

```
anc(Z,X) :- anc(Y,X), parent(Z,Y).
anc(Y,X) :- parent(Y,X).
```

Modify the program by reordering rules and literals to make it more efficient for the mode of usage in which the second argument is given. Give simplified derivation trees for the goal `anc(X,peter)` for the original and the reordered program.

7.4. The following program

```
 erk(X,Y)     :-  erk(X1,Y1), X1 = X - 1, Y = Y1 + 2.
 erk(0, 0).
```

has several problems. For example, the goal `erk(5, Y)` does not terminate.

(a) Modify the program by reordering rules and literals so that `erk(5, Y)` finds a first answer.

(b) Change the program so that the derivation tree for `erk(5, Y)` is finite.

(c) Change the program further so that the goal `erk(X, 11)` has a finite derivation tree.

(d) Can you write an equivalent program which does not use recursion?

7.5. Consider the `member` predicate on page 193, and the following alternate code for `member`.

```
member(X, [Y|R]) :- (X = Y -> true ; member(X, R)).
```

Draw the derivation trees for the goals

`member([p(kim,N),`*phonelist*`]).`

and

`member([p(P,5559282),`*phonelist*`]).`

for both codes for `member`. Explain the limitations of the above version of `member`.

7.6. "Stupid sort" is a sorting algorithm that sorts a list by trying all possible permutations of the list and checking whether the permutation is ordered. The predicate `perm(L,P)` defined below produces answers P which are all possible permutations of the list L. The predicate `delete(L,X,R)` deletes element X from list L leaving the remaining elements, R. The predicate `sorted(L)` checks if the list L of numbers is sorted.

```
stupid_sort(L, P) :- perm(L, P), sorted(P).

perm([], []).
perm(L, [X|R]) :- delete(L, X, L1), perm(L1, R).

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

sorted([]).
sorted([_]).
sorted([X,Y|L]) :- X ≤ Y, sorted([Y|L]).
```

Give the simplified derivation trees for the goals:

```
L = [2,6,2], P = [_,_,_], perm(L, P), sorted(P).

L = [2,6,2], P = [_,_,_], sorted(P), perm(L, P).
```

Compare the number of states. Both goals have two different derivations for the one answer. Using the derivation tree above, determine the number of states in the simplified derivation trees for the goal:

```
L = [2,6,2], P = [_,_,_], sorted(P), once(perm(L, P)).
```

7.7. Compare the execution of the goals `(` G_1 `->` G_2 `;` G_3 `)` and `( once(`G_1`)` `->` G_2 `;` G_3 `)` where G_1, G_2 and G_3 are arbitrary goals.

7.11 Practical Exercises

In $CLP(\mathcal{R})$ and SICStus Prolog `pow(`t_1`,` t_2`)` represents the expression $(t_1)^{(t_2)}$. For instance, $\sqrt{t}$ is `pow(`t`, 0.5)`.

P7.1. Test your reordered anc program from Question 7.3 above and compare its speed with the original version.

P7.2. Write a version of anc which is efficient for the mode of usage in which the first argument is fixed. Compare the speed with the reordered version of anc from Question 7.3.

P7.3. Write a program for `leaflevel_bst` which determines the level of the leaves in a binary search tree in which the items are positive numbers. Ensure that the goal `minimize(leaflevel_bst(T, X, N), X + N)` will execute efficiently, that is it will not always search the entire tree to find the minimum. [Hint: think about the goal `X + N <` m`, leaflevel_bst(T, X, N)`.]

P7.4. Write a version of `sumlist` which uses if-then-else and is correct if the first argument is fixed. Give a goal for which the behaviour of the program is incorrect.

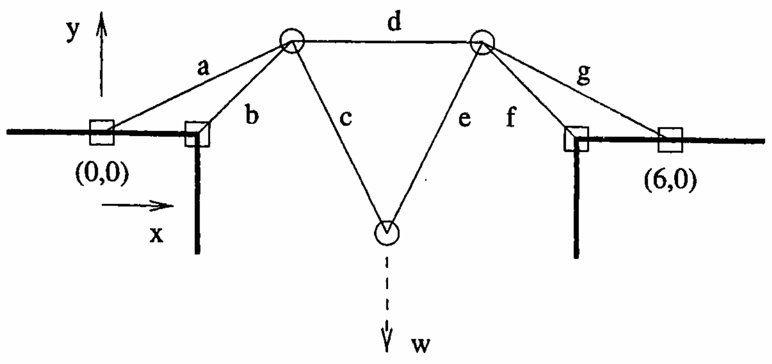


Figure 7.8 A different topology.

P7.5. Using the "stupid sort" program defined in Exercise 7.6 above, compare the execution time of the following goals to sort the list $[1, 6, 3, 4, 2, 6, 3, 2, 4]$:

```
L = [1,6,3,4,2,6,3,2,4], P = [_,_,_,_,_,_,_,_,_], perm(L,P), sorted(P).
L = [1,6,3,4,2,6,3,2,4], P = [_,_,_,_,_,_,_,_,_], sorted(P), perm(L,P).
L=[1,6,3,4,2,6,3,2,4],P=[_,_,_,_,_,_,_,_,_],sorted(P),once(perm(L,P)).
```

Compare the execution time when the constraint $0 = 1$ is appended on the end of the goal. This in effect searches for every solution and then fails.

P7.6. Write a predicate `strut_inter(S1,S2)` which holds if two struts (as defined in the bridge example) $S1$ and $S2$ intersect. Use it to write a predicate `not_strut_inter(S1,S2)` which holds if the two struts do not intersect. Determine the modes of usage for each predicate for which they will answer correctly.

P7.7. Write a rule for the `tpl` predicate for spaghetti bridges for the topology illustrated in Figure 7.8. Find the strongest bridge possible under this topology. Choose a different topology of your own making, and see if you can create a stronger bridge than that found in Section 7.8.2

P7.8. (*) The local search used to find a better solution to the bridge could be improved. If the solution is improved by perturbing by distance D, it may be improved further by perturbing again by distance D. Only if perturbing by distance D fails to improve the solution enough should we halve the perturbation length. Halt the process when the perturbation length becomes less than 0.05. Implement this search and see if you can find a stronger bridge than that discovered in Section 7.8.2 with the same topology.

P7.9. (*) Improve the spaghetti bridge analysis program so it will take into account the weight of the spaghetti, 100 g per metre for struts, and 0.5 g for each floating join. Assume it acts through the midpoint of the strut or join.

7.12 Notes

The discussion of efficiency and techniques for ordering rules and user-defined constraints in rules originates from Prolog programming and is described in Prolog programming texts such as that of Sterling and Shapiro [125].

Modes of usage are important for constraint logic programming languages because the same program can work for multiple modes of usage (for example the `mortgage` program of the Introduction). For *CLP*(*Tree*), that is Prolog, most arguments have a mode which is either fixed or free. For other constraint domains other modes of usage are more common. Arguments may be bounded from above, or below, or both, or constrained to be positive, or simply constrained in some unspecified way. In the modern logic programming language Mercury [122], modes of usage of predicates are explicitly defined by the programmer and checked by the compiler. This enables significantly more errors of usage to be discovered and allows the compiler to execute predicates more efficiently.

Guidelines for placing constraints in a rule and the use of answer redundant constraints were discussed by Marriott and Stuckey [92]. They also described how reordering and addition of redundant constraints could be automated.

If-then-else and once originated in Prolog, and their pitfalls are well known from Prolog folklore. In Prolog they are usually understood (and defined) using a more basic construct, the cut. We introduce the cut in Chapter 10 to explain the implementation of if-then-else and once. Naish's book [97] discusses in some detail, among many other things, modes of usage for Prolog and their interaction with if-then-else and cut.

Constraint logic programming has been used extensively for solving design problems. Some examples are: analog circuitry synthesis [118], transistor amplifier design [62], analysis and partial synthesis of truss structures [81], designing gear boxes [126] and other mechanical systems [130].