9 Advanced Programming Techniques

In the preceding chapters we have shown how the constraint programmer can use CLP to write programs which provide a high-level model of an application and which can be used to efficiently answer queries about the model. Sometimes, however, the underlying constraint solver or search mechanism of the CLP system may not be sufficiently powerful or expressive enough for the task at hand. One of the main strengths of CLP languages is that, as well as being modelling languages, they are also fully-fledged programming languages which can be used to extend the underlying constraint solver and to provide different search and minimization behaviour.

In this chapter, we look at several advanced programming techniques which the constraint programmer can employ to code sophisticated constraint solving and optimization algorithms within the CLP language itself. These include symbolic arithmetic reasoning using tree constraints, dynamic scheduling and meta-programming. Finally, we discuss the built-ins or library functions provided by most CLP languages since they are often used in conjunction with the techniques described in this chapter.

9.1 Extending the Constraint Solver

A CLP system provides a language for defining user-defined constraints and its evaluation mechanism a method for solving these user-defined constraints by rewriting them into primitive constraints which can be handled by the underlying solver. So, in a certain sense, all programming in a CLP language can be viewed as extending the underlying constraint solver.

However the majority of user-defined constraints are application specific and designed to work only for restricted modes of usage. This is in contrast to constraint solvers which support constraints that are applicable in many different contexts and which can be called in any mode of usage. Sometimes, however, it is useful for the programmer to define constraints which are sufficiently generic to be useful in many applications and whose implementation is robust enough to allow these constraints to be called with a variety of modes of usage. Such programming is said to extend the constraint solver. There is, of course, a continuum between application specific constraints and constraint solver extension so the difference is a question of degree

rather than absolute.

In this section we will investigate three natural extensions of the real arithmetic solver of $CLP(\mathcal{R})$ which illustrate this style of programming. In each case we take a constraint domain which is not supported by the built-in solver, choose a representation for elements in the constraint domain and then write predicates which define the primitive constraints and functions over the domain in terms of the underlying solver.

The first, and simplest, extension of the solver is to allow complex numbers. Indeed, we have already seen how to do this in Section 6.5. We can represent the complex number $x + iy$ by the record $c(x, y)$. Equations over complex numbers are provided by term equality "=." As we saw earlier, addition and multiplication of complex numbers is defined by

```
c_add(c(R1,I1),c(R2,I2),c(R3,I3)) :-
     R3 = R1 + R2, I3 = I1 + I2.
c_mult(c(R1,I1),c(R2,I2),c(R3,I3)) :-
     R3 = R1*R2 - I1*I2, I3 = R1*I2 + R2*I1.
```

Using these definitions we can model a variety of applications by means of complex numbers.

A slightly more difficult example is to extend the solver to handle vectors. We can represent a vector $\langle x_1, \ldots, x_n \rangle$ by the list $[x_1, \ldots, x_n]$. With this representation it is straightforward to write predicates `v_add`, `vs_mult`, `element` and `dotprod` which, respectively, add two vectors, multiply a vector by a scalar, access the i^{th} element in a vector and compute the dot product of two vectors. We give the code for vector addition and leave the code for the three other predicates as an exercise.

```
v_add([],[],[]).
v_add([X1|Y1], [X2|Y2], [X|Y])  :-  X = X1+X2, v_add(Y1,Y2,Y).
```

Again equations over vectors are provided for by term equality. An inequality relation over vectors is simply defined by

```
v_leq([],[]).
v_leq([X1|Y1], [X2|Y2])  :-  X1 ≤ X2, v_leq(Y1,Y2).
```

Our third example is to extend the solver to the sequence constraints domain described in Chapter 1. We can represent sequences using lists. As usual, equations over sequences are provided for by term equality. It is simple to write a function `concat` which concatenates a non-empty list of sequences to give a sequence and a predicate `not_empty` which holds if a sequence is non-empty.

```
not_empty([_|_]).
concat([S1],S1).
concat([S1,S2|Ss],S)  :-  append(S1,T,S), concat([S2|Ss],T).
```

The solution to a set of sequence constraints can be determined by using the `concat` predicate to find all possible concatenations through search and so we have a simple constraint solver for sequence constraints.

We can translate the contig problem discussed in Section 1.5.2 into a program over tree constraints. The problem is, given three sequences of chromosomes: a-t-c-g-g-g-c, a-a-a-a-t-c-g and g-c-c-a-t-t, find a relative order for which the sequences overlap to build a single sequence.

```
sequence_problem(T) :-
     contigs(Contigs),
     perm(Contigs, [C1, C2, C3]),
     not_empty(O12), not_empty(O23),
     concat([UC1,O12], C1),
     concat([O12,UC2,O23], C2),
     concat([O23,UC3], C3),
     concat([UC1,O12,UC2,O23,UC3], T).

contigs([[a,t,c,g,g,g,c], [a,a,a,a,t,c,g], [g,c,c,a,t,t]]).

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

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

The sequence constraint $X :: Y = Z$ is translated to the user-defined constraint `concat([X,Y],Z)`. Longer sequence concatenations, for example $T = UC1 :: O12 :: UC2 :: O23 :: UC3$ are also translated using `concat`. The constraint $X \neq \epsilon$, indicating X is not an empty sequence, is translated to `not_empty(X)`. The predicate `perm` generates all permutations of the first argument, by deleting some element from the list (using `delete`) and placing it in the front of a permutation of the remaining elements of the list. The `delete` predicate holds if the third argument is the list resulting from deleting the second argument from the first argument which is a list. The goal `sequence_problem(T)` finds the single answer

```
T = [a, a, a, a, t, c, g, g, g, c, c, a, t, t].
```

The goal works by first choosing a relative order of the contigs using `perm` and then checks the sequence constraints.

These examples also illustrate some of the limitations of programming constraint solver extensions in a CLP language. The first is that you cannot overload the function and primitive constraint names already used in the solver. Thus, we could not overload $\leq$ so that it also stood for vector inequality. Instead we had to introduce another name. The second limitation is that, in most CLP systems, it is not possible to directly define functions. Rather a function, such as addition, must be encoded as a relationship between its arguments and an extra argument which is the result of

applying the function to these arguments. This has the consequence that complex expressions must be flattened into a sequence of predicate calls by introducing temporary variables. For example, the vector constraint $\vec{x} = \vec{y} + 2 \times \vec{z}$ must be written by the programmer as

```
vs_mult(2,Z,Temp), v_add(Y,Temp,X).
```

These restrictions mean that programming over a constraint domain which is an extension of the underlying solver is more cumbersome than one in which the constraint solver directly supports the domain.

A more important limitation is that, because of the standard evaluation mechanism employed by the CLP system to process the constraints in the extension, often the constraints can only be used in a restricted mode of usage. For other modes of usage, the extended solver may not be well-behaved or, even worse, may not terminate.

In the contig ordering program the order of literals was carefully chosen. If we delay the choice of the ordering of the contigs until after the sequence constraints, the program would not determine an answer. The problem arises because the `concat` predicate provides only a weak constraint solver for concatenation constraints. If the first or second argument to `concat` is not fixed, evaluation may give rise to an infinite derivation. A simpler example of the same behaviour can be seen when trying to solve the (unsatisfiable) sequence constraint problem of finding two sequences $L1$ and $L2$, where $L2$ is not empty, which give the empty list when concatenated. This can be expressed as the goal

```
not_empty(L2), concat([L1,L2], L), L=[].
```

The goal runs forever without finding an answer because evaluation of the second "constraint" gives rise to an infinite search for answers. In effect, it tries $L1 = []$, then $L1 = [\_]$, $L1 = [\_,\_]$, and so on. None of which lead to an answer.

In Sections 9.6 and 9.7 we shall see techniques for modifying the default left-to-right evaluation mechanism of CLP languages. This ability allows us to program constraint solvers which are robust and which terminate in all modes of usage.

Another reason that we may wish to extend the underlying constraint solver is that the solver is not powerful enough to handle the constraints we wish to use, even if it is a solver for that constraint domain. This problem may arise, of course, whenever the underlying solver is incomplete.

For example, the $CLP(\mathcal{R})$ solver is incomplete for nonlinear arithmetic constraints. What can we do if our application requires us to solve nonlinear constraints? The answer is simple—write a solver in $CLP(\mathcal{R})$ for the particular nonlinear constraints we are interested in.

Imagine that we have a nonlinear arithmetic function f which has a single unknown variable x and we wish to solve $f(x) = 0$. For example, suppose $f(x)$ is $(x + 2)^3$. Evaluating the goal

```
(X + 2) * (X + 2) * (X + 2) = 0.
```

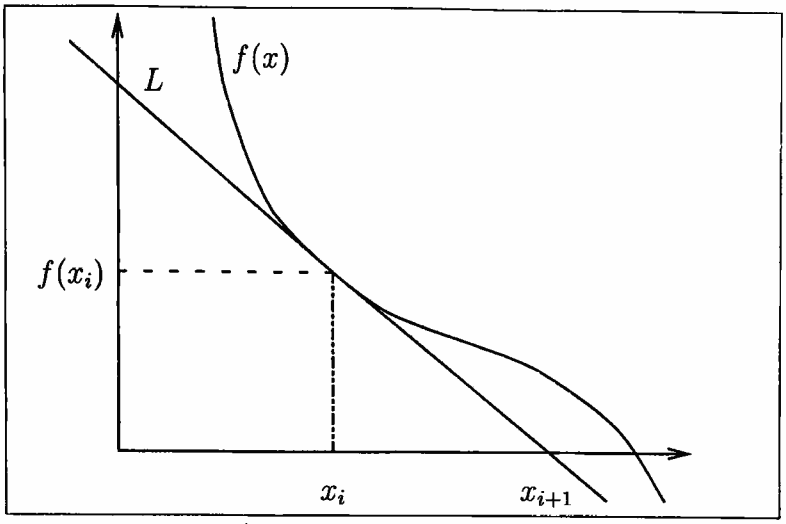

Figure 9.1 Newton-Raphson iteration.

with $CLP(\mathcal{R})$ will give the answer

```
0 = (X + 2) * (X + 2) * (X + 2)
*** Maybe
```

indicating that the $CLP(\mathcal{R})$ constraint solver cannot determine the satisfiability of the constraint since it is nonlinear.

However, there is no reason why we cannot program a more sophisticated constraint solver for nonlinears. CLP languages are true programming languages, and therefore it is possible to program any constraint solving algorithm in the language itself. Indeed, it is often very easy to write constraint solvers in a CLP language since the programmer can leverage from the underlying constraint solver.

To return to our example, given that the function f is differentiable, one of the simplest techniques for finding x such that $f(x) = 0$ is Newton-Raphson iteration. This starts from an initial guess at the solution x_0, and repeatedly computes a new guess, x_{i+1}, of the solution from the previous solution, x_i, as follows. The new solution is computed by approximating f by a line, L, which passes through the point $(x_i, f(x_i))$ and which has gradient $f'(x_i)$ where f' denotes the derivative of f. The value for x_{i+1} is simply the x-axis intercept of the line L. This is shown graphically in Figure 9.1. The process terminates when x_i is sufficiently close to a solution of $f(x) = 0$. That is, when $-\epsilon \leq f(x_i) \leq \epsilon$ for some given $\epsilon > 0$.

It is straightforward to write a program which uses Newton-Raphson iteration to solve this type of constraint. The program assumes that the relation $f(x) = y$ is defined by a predicate `f(x,y)` and $f'(x) = y$ is defined by a predicate `df(x,y)`.

Imagine that the function we wish to solve is $(X + 2)^3$. This has derivative, $3 * (X + 2)^2$. The program is simply

```
f(X, F)              :-  F = (X+2)*(X+2)*(X+2).
df(X,DF)             :-  DF = 3*(X+2)*(X+2).

solve_nr(E, X0, X0)  :-  f(X0,F0), -E ≤ F0, F0 ≤ E.
solve_nr(E, X0, X)   :-  f(X0, F0),
                         df(X0, DF0),
                         F0 = DF0 * X0 + C,
                         0 = DF0 * X1 + C,
                         solve_nr(E, X1, X).
```

The goal `solve_nr(E, X0, X)` finds a value for X which satisfies

$$-E \leq f(X) \leq E$$

by using Newton-Raphson iteration and starting from the initial guess $X0$. The first rule simply checks if the current guess is sufficiently close to the solution. If so the process terminates. Otherwise the second rule computes a new guess at the solution $X1$ as the point where the line L, defined by $y = DF0 \times x + C$, hits the x axis.

The intended mode of usage for `solve_nr` is that the second argument is fixed, so the nonlinear constraints encountered in `f` and `df` are effectively linear and are therefore completely handled by the solver. Subsequent solutions will converge toward the true zero.

Given the goal `solve_nr(0.0001,0.0,X)`, the first answer found is $X = -1.96532$. The second answer is $X = -1.97688$. The goal actually has an infinite sequence of answers that approach the actual answer -2. Since the first answer is an answer that meets the accuracy required, a sensible goal for usage of this program is instead `once(solve_nr(0.0001,0.0,X))`, which only has a single answer, $X = -1.96532$. For this mode of usage in which the first argument is also fixed, it is possible to write a deterministic version of the program that uses if-then-else.

As it is written, the predicate `solve_nr` has two defects. First, the function whose zero is to be found is hardwired into the predicate. Second, the programmer has had to explicitly provide the derivative, rather than letting the program itself compute it from the function. In the next section we develop a better version of the program which overcomes both shortcomings.

9.2 Combined Symbolic and Arithmetic Reasoning

A natural strength of CLP languages which provide tree constraints and arithmetic constraints, is the ability to combine symbolic and arithmetic reasoning. Tree constraints may be used for symbolic reasoning, while arithmetic constraints may be used for arithmetic reasoning. Combining both types of reasoning can lead to sophisticated and powerful programs.

In this section we will modify the above program for finding a zero for a nonlinear function so that the function and its derivative are not hardwired into it. We do this by adding a parameter which is the function whose zero is to be computed. We represent this function by a tree and the program can "symbolically" differentiate the function to find the derivative.

We can represent mathematical expressions over a variable x as follows. A number is represented by itself, the variable of interest is simply the constant x, and the trees $plus(t_1, t_2)$, $mult(t_1, t_2)$, and $power(t_1, t_2)$ represent the expressions $t_1 + t_2$, $t_1 \times t_2$, and $(t_1)^{(t_2)}$, respectively. Similarly, we use the trees $minus(t)$, $sine(t)$ and $cosine(t)$ to represent the expressions $-t$, $sin(t)$ and $cos(t)$ respectively.

For instance, the function over x defined by the expression $x^2 + 3x + 2$ may be represented by the tree $plus(power(x, 2), plus(mult(3, x), 2))$.

Given a tree representation for a function over x, we can write a predicate which returns the value of the function for a given value of x. The user-defined constraint `evaln(T, X, V)` does this, returning in V the value of the function represented by the tree T for the value of x given in X.

```
evaln(x,X,X).
evaln(N,_,N) :- arithmetic(N).
evaln(power(x,N),X,E) :- E = pow(X, N).
evaln(sine(x),X,E) :- E = sin(X).
evaln(cosine(x),X,E) :- E = cos(X).
evaln(minus(F),X,E) :- E = -EF, evaln(F,X,EF).
evaln(plus(F,G),X,E) :- E = EF + EG, evaln(F,X,EF), evaln(G,X,EG).
evaln(mult(F,G),X,E) :- E = EF * EG, evaln(F,X,EF), evaln(G,X,EG).
```

The program makes use of the library function `arithmetic(X)` which succeeds if X is determined by the solver to take a fixed arithmetic value. The predicate `evaln` recursively traverses the sub-expressions in tree T, evaluating each sub-expression for the given value of X, and then appropriately combines the results. For instance, `evaln(plus(power(x,2),plus(mult(3,x),2)),2,V)` returns the answer $V = 12$.

However, so long as the first argument is fixed, this program can be used in other modes of usage. The goal `evaln(T, X, V)` adds constraints which ensure that V is constrained with respect to X analogously to the way expression T is constrained with respect to x. For instance, the goal

```
evaln(plus(power(x,2),plus(mult(3,x),2)),X,F).
```

succeeds with a single answer which constrains F to be equal to $X^2 + 3X + 2$.

The main interest in having a symbolic representation of an arithmetic expression is that it allows us to manipulate the expression symbolically. One useful symbolic manipulation of an arithmetic expression over variable x is to differentiate it with respect to x.

Given the goal `deriv`(F, DF), the following program takes the expression F over x and computes its derivative DF with respect to x. It is designed to work deterministically in the mode of usage where the first argument is fixed.

```
deriv(x,1).
deriv(N,0) :- arithmetic(N).
deriv(power(x,N),mult(N,power(x,N1))) :- N1=N-1.
deriv(sine(x),cosine(x)).
deriv(cosine(x),minus(sine(x))).
deriv(minus(F),minus(DF)) :- deriv(F,DF).
deriv(plus(F,G),plus(DF,DG)) :- deriv(F,DF), deriv(G,DG).
deriv(mult(F,G),plus(mult(DF,G),mult(DG,F))) :-
     deriv(F,DF), deriv(G,DG).
```

For instance, the goal `deriv(plus(power(x,2),plus(mult(3,x),2)),F)` succeeds with answer

$$F = plus(mult(2, power(x, 1)), plus(plus(mult(0, x), mult(1, 3)), 0))$$

which represents the expression $2x^1 + 0x + 1 \times 3 + 0$ or, equivalently, $2x + 3$.

The present program cannot handle all such expressions, for example it cannot differentiate $sin(cos(x))$, nor, as we have seen, does it perform simplification. It is left as an exercise to add these features.

Using these two rather simple user-defined constraints, we can now build an improved Newton-Raphson solver which makes use of symbolic reasoning to evaluate and differentiate the function whose zero we wish to find.

The program needs to be only slightly modified. We need arguments, say F and DF, to carry around the representations of the function and its derivative. These are used with `evaln` to determine values of the function and its derivative. The remainder of the program is the same. For the initial goal `dsolve` we use `deriv` to compute the derivative function. Thus the program is:

```
dsolve(E,F,X0,X) :- deriv(F,DF), solve_nr(E,F,DF,X0,X).

solve_nr(E,F,_DF,X0,X0) :- evaln(F,X0,F0), -E ≤ F0, F0 ≤ E.
solve_nr(E,F,DF,X0,X) :-
     evaln(F,X0,F0),
     evaln(DF,X0,DF0),
     F0 = DF0 * X0 + C,
     0 = DF0 * X1 + C,
     solve_nr(E,F,DF,X1,X).
```

For example given the function on x described by the expression $x^2 + 3x + 2$ we can determine a zero of the expression starting from $x = 5$ and within accuracy 0.00001 by the following goal:

```
dsolve(0.00001, plus(power(x,2), plus(mult(3,x), 2)), 5, X).
```

The answer is $X = -1$.

9.3 Programming Optimization

In the last two sections we have seen how to extend the underlying constraint solver by either supporting constraints over a new constraint domain or by building a solver which is more powerful than that provided by the CLP system for specialized forms of constraints. However, testing constraint satisfaction is only one of the facilities provided by a CLP system.

Another important function provided by the CLP system is optimization, that is, finding a solution which minimizes a particular expression. Just as a CLP programmer can extend the constraint solver, they can extend or modify the optimization facilities provided by their system. We now investigate how to do this by means of two examples.

The first example illustrates how to extend the minimization facilities provided by a real arithmetic constraint solver in order to handle integer constraints. The technique used is branch and bound. The second example shows how to develop a special purpose minimization predicate for a problem in which the programmer can make use of special properties of the problem to reduce the search space.

9.3.1 Branch and Bound

In Section 3.6 we gave a procedure for finding the optimal solution for an integer problem which is based on repeatedly solving the optimization problem over the real numbers. In essence the process is this. First, the optimal solution of the problem over the real numbers is found. If all values in this solution are integers, the problem is finished. Otherwise, there is a variable x whose value d in the optimal solution over the reals is not integral. Two new problems are generated with $x \leq \lfloor d \rfloor$ and $x \geq \lceil d \rceil$. Each of these is treated in turn using the method above. Eventually a solution containing only integer values will be discovered (assuming the solution space is bounded). The current best solution over the integers is threaded throughout the computation so that, if for any sub-problem the optimal solution over the reals is worse than the best current solution over the integers, the sub-problem is discarded since it can never yield a better solution over the integers.

This algorithm is straightforward to program in any CLP system providing real arithmetic optimization. We assume the optimization problem is defined by a predicate `problem(`*vars*`, f)` where *vars* is a list of the problem variables, and f is the optimization function. Since the optimization problem will be augmented with constraints placing upper and lower bounds on the variables, we need some structure in which to store information about these bounds. In the program below, we use a list of pairs of the form $b(lb, ub)$ where lb is either the constant u, indicating no lower bound (unbounded), or an integer, and ub is similarly defined. The n^{th} element of the list of pairs gives the bounds for the n^{th} element of the list *vars*.

The predicate bounds converts the bounds data structure in its second argument into constraints on the variables in the first argument. The predicate

bounded_problem constructs the constraints for the problem with the added bounds, while new_bounds takes an optimal solution over the reals to the problem, Vs, and the current bounds, $Bnds$, and builds two new bounds structures, $BndsL$ and $BndsR$, by splitting the current bounds on the first non-integral value in the solution. If the solution is completely integral then $BndsL = BndsR$.

These are called by the branch-and-bound predicate bnb. This takes the current minimal integral solution, $CVals$, and the current minimal optimization function value, $CBest$, together with the current bounds structures, $Bnds$, which defines the subspace of the original problem's solutions to be explored. It returns the best integer solution found after this solution subspace has been explored, $BestVals$, together with the corresponding value, $Best$, of the optimization function.

The predicate bnb first finds an optimal solution over the reals to the bounded problem, which must be better than the current best integer solution. If none exists, then the current best solution is returned. Otherwise the solution, Vs, is used to split the current bounds structure, $Bnds$, into two parts. If the solution, Vs, is all integral, that is $BndsL = BndsR$, this solution is returned as the best. Otherwise the two sub-problems are investigated in turn.

```
bnb(CBest, CVals, Bnds, Best, BestVals) :-
     (minimize((F < CBest, bounded_problem(Vs, Bnds, F)), F) ->
          new_bounds(Vs, Bnds, BndsL, BndsR),
          (BndsL = BndsR -> Best = F, BestVals = Vs
          ;
               bnb(CBest, CVals, BndsL, LBest, LVals),
               bnb(LBest, LVals, BndsR, Best, BestVals)
          )
     ;
          Best = CBest, BestVals = CVals
     ).

new_bounds([], [], [], []).
new_bounds([V|Vs], [B|Bs], [BL|BLs], [BR|BRs]) :-
     (integer(V) ->
          BL = B, BR = B,
          new_bounds(Vs, Bs, BLs, BRs)
     ;
          B = b(L,U),
          VL = ⌊ V ⌋,
          VU = ⌈ V ⌉,
          BL = b(L,VL), BR = b(VU, U),
          BLs = Bs, BRs = Bs
     ).
```

```
bounded_problem(Vs, Bnds, F) :-
     bounds(Vs, Bnds),
     problem(Vs, F).

bounds([], []).
bounds([V|Vs], [b(L,U)|Bs]) :-
     (L = u -> true ; L ≤ V),
     (U = u -> true ; V ≤ U),
     bounds(Vs, Bs).
```

The program makes use of the library function `integer(X)` which succeeds whenever X is constrained to take an integral value, as well as the functions $\lfloor V \rfloor$ and $\lceil V \rceil$ which return the largest integer no greater than V and the smallest integer no less than V respectively.

Example 9.1

Consider the simple integer optimization problem which is to maximize Y subject to the constraints

$$2Y \leq 3X - 3 \wedge X + Y \leq 5.$$

We can encode this as the minimization problem

```
problem([X,Y], -Y) :- 2 * Y ≤ 3 * X - 3, X + Y ≤ 5.
```

Figure 9.2 shows the solution space of this problem, together with the various subspaces that are explored by the predicate.

The initial call is

```
bnb(CBest1, CBestVals1, [b(u,u), b(u,u)], Best1, BestVals1).
```

The solid lines correspond to the initial inequalities. Minimizing `bounded_problem` finds the optimal solution of the initial real problem is $X = 2.6 \wedge Y = 2.4$ (indicated by a small box). Execution of `new_bounds` finds that X is the first variable with a non-integer solution. It sets $BndsL$ to $[b(u,2), b(u,u)]$ and $BndsR$ to $[b(3,u), b(u,u)]$, effectively splitting the solution space by using the constraints $X \leq 2$ and $X \geq 3$. These inequalities are shown as dashed vertical lines in the diagram.

Computation continues with the call

```
bnb(CBest1, CBestVals1, [b(u,2), b(u,u)], Best2, BestVals2),
```

which investigates the left subspace of solutions. Minimizing `bounded_problem` finds the optimal solution $X = 2 \wedge Y = 1.5$. The call to `new_bounds` finds that Y is the first variable with a non-integer solution and sets $BndsL$ to $[b(u,2), b(u,1)]$ and $BndsR$ to $[b(u,2), b(2,u)]$. The split of the solution space is shown by dashed horizontal lines in the diagram.

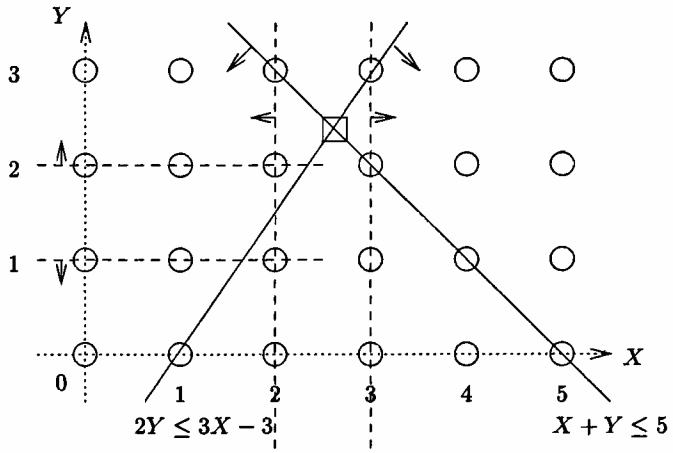

Figure 9.2 Diagram of (X,Y) space for branch and bound optimization.

Computation continues with the call

`bnb(CBest`$_1$`, CBestVals`$_1$`, [b(u,2), b(u,1)], Best`$_3$`, BestVals`$_3$`).`

The optimal solution found is $X = 2 \wedge Y = 1$. Since this solution is integer, new_bounds constructs $BndsL = BndsR = [b(u,2), b(u,1)]$ and so the call returns $Best_3 = 1$ and $BestVals_3 = [2,1]$.

The next call is the second half of the split on Y with $CBest$ given by $Best_3$ and $CBestVals$ given by $BestVals_3$. The call is

`bnb(1, [2,1], [b(u,2), b(2,u)], Best`$_2$`, BestVals`$_2$`).`

The goal

```
F < 1, bounded_problem(Vs, [b(u,2), b(2,u)], F)
```

fails and therefore the minimization goal also fails. Thus $Best_2$ becomes 1 and $BestVals_2$ becomes [2,1].

The remaining part of the computation is the second half of the split on X with $CBest$ given by $Best_2$ and $CBestVals$ given by $BestVals_2$. The call is

`bnb(1, [2,1], [b(3,u), b(u,u)], Best`$_1$`, BestVals`$_1$`).`

The minimization goal returns solution $X = 3 \wedge Y = 2$, and $Best_1$ becomes 2 and $BestVals_1$ becomes [3,2]. This is the answer returned for the original goal.

Example 9.2

Consider the integer optimization problem for the smuggler's knapsack defined in Example 3.7 in which we wish to maximize $15W + 10P + 7C$ subject to

$$4W + 3P + 2C \leq 9 \wedge 15W + 10P + 7 \geq 30 \wedge CW \geq 0 \wedge P \geq 0 \wedge C \geq 0.$$

```
problem([W,P,C], -Profit) :-
     4 * W + 3 * P + 2 * C ≤ 9,
     W ≥ 0, P ≥ 0, C ≥ 0,
     Profit = 15 * W + 10 * P + 7 * C, Profit ≥ 30.
```

The goal

```
bnb(_, _, [b(u,u),b(u,u),b(u,u)], Best, BestVals).
```

returns the answer

$$Best = -32 \wedge BestVals = [1, 1, 1].$$

indicating that the most profit of 32 (minimum loss) is obtained by taking one of each item.

9.3.2 Optimistic Partitioning

The above program illustrates how to define a minimization function over a new constraint domain by leveraging from the underlying minimization predicate provided by the CLP system. However, even if the CLP system provides a minimization operator over the constraint domain of interest, it is sometimes possible for the CLP programmer to write a problem-specific minimization predicate which is more efficient than that provided by the CLP system by utilizing knowledge about the structure of a problem's solution space.

In Section 8.5 we used the built-in search predicate `minimize` to find an optimal solution to the original code of the simple scheduling program (appearing on page 273) using the goal:

```
minimize((schedule(problem, End, Joblist), indomain(End)), End).
```

where *problem* is the simple problem illustrated in Figure 8.3.

The typical implementation of `minimize` works by repeatedly finding a solution to the goal which is better than the current minimum. This process is similar to that employed in the backtracking integer optimizer back_int_opt. For this example, evaluation proceeds by finding the first answer

$$TS_1 = 8 \wedge TS_2 = 0 \wedge TS_3 = 24 \wedge TS_4 = 18 \wedge TS_5 = 15 \wedge TS_6 = 11$$

with $End = 32$. The minimizing routine continues by adding $End \leq 31$ as a constraint, and finding an answer to this new problem. The answer found is

$$TS_1 = 8 \wedge TS_2 = 0 \wedge TS_3 = 20 \wedge TS_4 = 14 \wedge TS_5 = 11 \wedge TS_6 = 20$$

with $End = 28$. Computation continues searching with $End \leq 27$. Solutions are found for $End = 26$, $End = 22$, $End = 20$ and finally the optimum in which $End = 19$. The minimization routine continues to search for a solution with

$End \leq 18$ but this fails.

This example illustrates that for this problem the search for an optimal solution involves computing a long sequence of answers, each slightly better than the last, until the optimal solution is eventually reached. In a sense, the default minimization routine performs a linear search through the solution space in which solutions are ordered by the best value of *End* they allow. Better search strategies are possible.

One alternative search strategy is optimistic partitioning. This works as follows. Given an upper bound, Max, and lower bound, Min, for the value of the objective function, *End*, we may hope to reduce the number of solutions considered by performing a binary search through the solution space in a way analogous to domain splitting. We first search for a solution in the lower half of the range. If this is found we then look for a better solution in the lower half of the remaining range. If there is no solution in the lower half of the range, we search in the upper half of the range. At each step in the search we keep track of the minimum and maximum values that *End* can take. The program below implements this search strategy for minimization.

```
split_min(Data, Min, Max, Joblist0, Joblist) :-
     Mid = (Min + Max) div 2,
     (End ≤ Mid, End ≥ Min,
      schedule(Data, End, Joblist1), indomain(End) ->
          NewMax = End - 1,
          split_min(Data, Min, NewMax, Joblist1, Joblist)
     ; (End ≤ Max, End ≥ Mid + 1,
      schedule(Data, End, Joblist1), indomain(End) ->
          NewMin = Mid + 1, NewMax = End - 1,
          split_min(Data, NewMin, NewMax, Joblist1, Joblist)
     ; Joblist = Joblist0
     )).
```

Given a current minimum, Min, and maximum, Max, on the objective function, as well as a current best solution, *Joblist*0, (encoded as a joblist), `split_min` determines the midpoint of the current range, *Mid*, and tries to find a solution in the lower half first. If this succeeds and finds a new solution, *Joblist*1, (implicitly the if-then-else only finds a single solution) we continue by updating the maximum to be less than the new solution and updating the current best solution to *Joblist*1. If this fails, we search for a solution in the upper half of the range, and, if we find one, continue as above. If no solution is found in both ranges then the current best solution is returned since it is still the best solution found so far.

For the project described by *problem* we can reason as follows. For each machine, the sum of the durations of tasks that require this machine is a lower bound for the total time. So, in this case 19 and 13 are lower bounds on the total time. The sum of the durations of all tasks is clearly an upper bound, in this case 32. Therefore, we ask the initial goal

split_min(*problem*, 19, 32, [], Joblist).

The goal executes as follows: *Mid* is calculated to be 25 and execution searches for a schedule with *End* time between 19 and 25. A solution, sol_1, is found with $End = 22$. Next the call

split_min(*problem*, 19, 21, sol_1, Joblist)

is evaluated. *Mid* is calculated to be 20 and a solution, sol_2, is found with $End = 20$. In turn, the call

split_min(*problem*, 19, 19, sol_2, Joblist)

is executed. Now a solution, sol_3, is found at $End = 19$ (the optimal) and the call

split_min(*problem*, 19, 18, sol_3, Joblist)

is executed, which immediately sets *Joblist* to sol_3.

The ability to define and program specific optimization strategies, such as optimistic partitioning, is a major strength of constraint logic programming. For this example problem, the cost of continually resolving the scheduling problem outweighs the advantage of finding less solutions before the the optimal solution is reached, so optimistic partitioning is not worthwhile. However, optimistic partitioning is advantageous if the search space of the goal to be minimized when given an upper and lower bound, that is to say, $l \leq$ F, F $\leq u$, goal(F), is substantially smaller than the search space for goal(F). The strategy will find an optimal solution after discovering a chain of solutions of length at most logarithmic in the size of the initial range $Max - Min$.

9.4 Higher-order Predicates

Most CLP predicates take expressions constructed from variables and the functions and constants of the underlying constraint domain. We have, however, also met built-in predicates which take a goal as an argument, for example, minimize, once and if-then-else literals. Since they take a constraint or goal as an argument, such predicates are said to be *higher-order*.

Higher-order predicates are not only provided by the library of the CLP system. The constraint programmer is also free to define their own. This is surprisingly easy. Goals and constraints which are passed as arguments to a higher-order predicate are simply treated as if they were terms in which predicate symbols and the literal conjunction operator "," are treated as tree constructors.

For example, consider the call

once((member(X,L1), X = Y, member(Y,L2)))

which determines if lists $L1$ and $L2$ share a common element. The extra parenthesis are required to ensure the goal

member(X,L1), X = Y, member(Y,L2).

is not treated as three arguments.

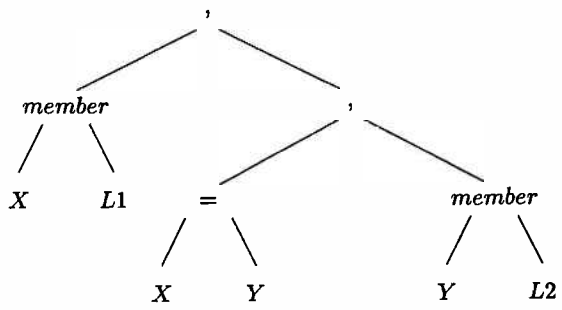

Figure 9.3 A term describing the argument to `once`.

The argument to `once` is treated as the term shown in Figure 9.3. Note how "," in the argument to `once` (but not inside the `member` literals) is treated as a binary tree constructor with the goals it conjoins as its arguments.

This means that programming with higher-order arguments is essentially like programming with trees. The only difference is, sometimes, we wish to evaluate a goal or literal which has been passed as an argument. This may be done by using `call`, the most fundamental higher-order built-in. The predicate `call` takes a goal as its single argument and returns the result of evaluating the goal. If evaluation of the goal finitely fails, the call to `call` finitely fails. In other words, the goal `G` behaves identically to the literal `call(G)`.

When using `call`, the programmer must be careful to ensure that, at the time of evaluation, its argument is constrained to be a term with the syntax of a goal. Otherwise evaluation will cause a *run-time* error because the system has no idea of how to evaluate it. Thus, the goals `call(Y)` and `call((p(X),Y))` will cause run-time errors since the variable Y is not instantiated to a literal or a goal. On the other hand, the goal `X=p(Y), call(X)` will return the answers to the literal `p(Y)`.

As a simple example of higher-order programming, imagine that our CLP system does not provide `once` but does provide if-then-else. It is simple to define `once` in terms of if-then-else:

```
once(G) :- (call(G) -> true; fail).
```

Here, for the first time in a program, we use the built-in literal `fail`. The fail literal, `fail`, is equivalent to an unsatisfiable constraint, say $0 = 1$. It causes failure when it is rewritten. It is used here to ensure failure when the once goal has no solutions.

This definition of `once` will have exactly the same behaviour as the usual built-in. It first uses `call` to evaluate the goal G. If evaluation of G succeeds, it returns this answer, otherwise, it finitely fails. On backtracking it will not consider other answers to G.

In a well designed CLP system there is little need for the programmer to define higher-order predicates as almost any predicate needed will be provided as a built-in. However, occasionally it is useful, and a knowledge of higher-order programming allows the programmer to implement built-ins missing from their system.

9.5 Negation

One important construct missing from the discussion of modelling with CLP languages is negation. If the constraint domain is rich enough, the negation of every primitive constraint is also a constraint. This holds for the finite domain constraints since we have $=$ and $\neq$ which are clearly the negation of each other, and $\leq$ has as its negation $>$.

Using complementary primitive constraints it is often possible for the programmer to explicitly define the negation of a user-defined constraint. For instance, recall the user-defined constraint `member(X, L)` which holds if X is a member of the list L. We can define its negation using $\neq$ as follows:

```
not_member(_X,[]).
not_member(X,[Y|L]) :- X ≠ Y, not_member(X,L).
```

Indeed, we already saw this user-defined constraint in Section 6.2.

However, writing such user-defined constraints is tedious and sometimes difficult, especially if the underlying solver does not provide the negation of all primitive constraints. For this reason most CLP languages provide a generic form of negation, by means of the higher-order built-in `not`, which can be applied to any goal G to give its negation `not(G)`. The drawback of `not` is that it only behaves correctly for some modes of usage. With other modes, its behaviour may be counter-intuitive, so the programmer needs to be aware of the way in which it will be called.

Definition 9.1
A *negative literal* is of the form $not(G)$ where G is a goal called the *negative subgoal.*

Evaluation of a negative literal is reasonably intuitive: when the literal `not(G)` is encountered, G is run to see if it finitely fails. If G finitely fails, then the negation of G holds so `not(G)` succeeds. Conversely, if G succeeds then, for some solution to the current store, the negation of G is not true, so `not(G)` fails. We extend the standard CLP evaluation mechanism by adding the following derivation step to process negative literals.

Definition 9.2
A *negation derivation step* is a derivation step $\langle G_1 \mid C_1 \rangle \Rightarrow \langle G_2 \mid C_2 \rangle$, in which a negative literal is rewritten as follows:
Let G_0 be the sequence of literals

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

and let L_1 be of the form $not(G)$. There are two cases.

1. If the state $\langle G \mid C_1 \rangle$ finitely fails, then C_2 is C_1 and G_2 is $L_2, \ldots, L_m$.
2. Otherwise, if evaluation of the state $\langle G \mid C_1 \rangle$ finds a successful derivation, then C_2 is $false$ and G_2 is the empty goal (■).

We explore the implementation of not further in Chapter 10.

One use of negative literals is to extend the power of the underlying constraint solver. Imagine that our underlying constraint solver supports equations but does not support disequations. We can use a negative literal to define disequality:

```
ne(X,Y)  :-  not(X=Y).
```

Consider the goal

```
X = 2, Y = 3, ne(X,Y).
```

Evaluation proceeds as follows. First `X = 2` and `Y = 3` are processed, giving the state

$$\langle ne(X,Y) \mid X = 2 \wedge Y = 3 \rangle.$$

The ne literal is rewritten to obtain the simplified state

$$\langle not(X = Y) \mid X = 2 \wedge Y = 3 \rangle.$$

Now the negative literal $not(X = Y)$ is processed. This spawns a sub-computation to evaluate the state $\langle X = Y \mid X = 2 \wedge Y = 3 \rangle$. This sub-computation finitely fails since the constraint $X = 2 \wedge Y = 3 \wedge X = Y$ is unsatisfiable. Thus the negative literal $not(X = Y)$ succeeds and so $\langle not(X = Y) \mid X = 2 \wedge Y = 3 \rangle$ is rewritten to $\langle \square \mid X = 2 \wedge Y = 3 \rangle$. Thus the initial goal succeeds with the single answer $X = 2 \wedge Y = 3$.

Consider the goal

```
X = 2, Y = 2, ne(X,Y).
```

Evaluation proceeds similarly to before, arriving at the state

$$\langle not(X = Y) \mid X = 2 \wedge Y = 2 \rangle.$$

The sub-computation to evaluate the state $\langle X = Y \mid X = 2 \wedge Y = 2 \rangle$ succeeds with answer $\langle \square \mid X = 2 \wedge Y = 2 \wedge X = Y \rangle$. Thus the negative literal $not(X = Y)$ fails and the state $\langle not(X = Y) \mid X = 2 \wedge Y = 2 \rangle$ is rewritten to $\langle \blacksquare \mid false \rangle$.

Unfortunately, because of the limitations of not, this predicate is only correct for the mode of usage in which both arguments have fixed values when it is called. For instance, the goal

```
X = 2, Y = 3, ne(X,Y).
```

will correctly succeed while the goal

```
ne(X,Y), X = 2, Y = 3.
```

will incorrectly fail. The state $\langle not(X = Y), X = 2, Y = 3 \mid true \rangle$ is reached. The sub-computation for the state $\langle X = Y \mid true \rangle$ clearly succeeds, and hence the whole derivation fails. Nonetheless, it does allow the programmer to use disequations,

although in a very restricted fashion. In the next section we shall see how to implement ne in a more general fashion.

Negative literals can only be guaranteed to act as one would expect when all the variables in the literal are determined by the constraint solver to take fixed values. This means that, when using an incomplete solver, the programmer must check that the solver is strong enough to infer the arguments to the negative literal are determined. For instance, the $CLP(\mathcal{R})$ system will finitely fail with the goal

```
Y * Y = 4, Y ≥ 0, not (Y ≥ 1).
```

A related problem with negative literals and incomplete solvers is that the solver may not be powerful enough to determine that the sub-computation spawned by the negative literal should finitely fail. The problem is that if the negative subgoal has a derivation in which the solver determines the satisfiability of the final constraint store is *unknown*, while the constraint is actually unsatisfiable, then the subgoal does not finitely fail, and so the negative literal will, erroneously, fail.

Consider the goal

```
X ≤ 0, Y ≥ 1, Z ≥ 2, not(X = Y * Z).
```

The goal fails because rewriting of literal `not(X = Y * Z)` requires the evaluation of the state

$$\langle X = Y \times Z \mid X \leq 0 \wedge Y \geq 1 \wedge Z \geq 2 \rangle.$$

This succeeds (in one step) with state

$$\langle \square \mid X \leq 0 \wedge Y \geq 1 \wedge Z \geq 2 \wedge X = Y \times Z \rangle,$$

since the constraint solver is incomplete for nonlinear constraints and cannot determine that this constraint is, in fact, unsatisfiable. Thus, even though there is an answer, the negative literal fails and so does the original goal.

Because of these problems, negative literals are rarely used in CLP programs for modelling the negation of user-defined constraints. Rather, as we saw earlier for `not_member`, the programmer must explicitly write a definition for the negation of the constraint. variables. are

There are, however, a number of situations in which negative literals are useful. The first situation is for data structure manipulation. Often when data structures are manipulated they have fixed values, meaning that they can safely be used as arguments to negative literals. For instance, if we were manipulating a fixed property list and wished to determine if an item with a fixed key was not in the list, then we could use a negative literal.

The second situation in which negative literals are useful is to test whether a constraint is compatible with the current store without actually adding the constraint to the store. Consider the user-defined higher-order predicate

```
is_compatible(G) :- not(not(G)).
```

Evaluation of `is_compatible(G)` will succeed if `not(G)` finitely fails. This occurs precisely when the subgoal G is compatible with the current constraint. However, since successful evaluation of a negative literal does not modify the constraint store, evaluation of `is_compatible(G)` will not change the store.

The constraint `is_compatible` is particularly useful when we wish to test whether a non-deterministic user-defined constraint is compatible with the store but do not want to commit to some choice in the definition of the user-defined constraint.

Example 9.3

Suppose we have a complicated problem, in which, midway through the evaluation, we wish to test whether variable X can still take a value whose absolute value is at least 4, because if it cannot, we know there is no solution. We might program this as

```
goal1(X, Y), Z ≥ 4, abs(X,Z), goal2(X, Y).
```

where

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

The disadvantage of this approach is that by using the `abs` goal the derivation will choose one possibility: either $X \geq 4$ or $X \leq -4$. Suppose that eventually in the evaluation of `goal2(X, Y)`, X is forced to be smaller than 4, then the execution will have to backtrack all the way to the second choice for `abs`. This may double the size of the derivation tree.

Instead we can use the goal

```
goal1(X, Y), is_compatible((Z ≥ 4, abs(X,Z))), goal2(X, Y).
```

This checks whether X can take a value whose absolute value is 4 or more without adding any constraints or setting up a choice point.

For simplicity, suppose that the subgoals are defined as follows.

```
goal1(X, Y) :- Y ≥ 1, X + Y = 6.
goal2(_X, Y) :- member(Y, [-21,-18,-10,-2,10,16,21,25]).
```

Note that given $X + Y = 6$, then `goal2` can only succeed if the absolute value of X is greater than 4. Now consider the leftmost successful derivation for the above goal, skipping some derivation steps. Note that we use $\Rightarrow^*$ to indicate several (collapsed)

derivation steps.

$$\langle goal1(X,Y), is\_compatible(Z \geq 4, abs(X,Z)), goal2(X,Y) \mid true\rangle$$
$$\Downarrow_*$$
$$\langle not(not(Z \geq 4, abs(X,Z))), goal2(X,Y) \mid Y \geq 1 \wedge X + Y = 6\rangle$$
$$\Downarrow_*$$
$$\langle goal2(X,Y) \mid Y \geq 1 \wedge X + Y = 6\rangle$$
$$\Downarrow_*$$
$$\langle \square \mid Y \geq 1 \wedge X + Y = 6 \wedge Y = 10\rangle$$

The sub-derivation for the state $\langle not(Z \geq 4, abs(X,Z)) \mid Y \geq 1 \wedge X + Y = 6\rangle$ is

$$\langle not(Z \geq 4, abs(X,Z)) \mid Y \geq 1 \wedge X + Y = 6\rangle$$
$$\Downarrow_*$$
$$\langle \blacksquare \mid false\rangle$$

The sub-derivation for the state $\langle Z \geq 4, abs(X,Z) \mid Y \geq 1 \wedge X + Y = 6\rangle$ is

$$\langle Z \geq 4, abs(X,Z) \mid Y \geq 1 \wedge X + Y = 6\rangle$$
$$\Downarrow_*$$
$$\langle \square \mid Y \geq 1 \wedge X + Y = 6 \wedge Z \geq 4 \wedge X = Z\rangle$$

The first answer for the original goal is found after the `member` literal tries setting Y to -21, -18, -10, and -2.

Now suppose the original goal was

```
goal1(X, Y), Z ≥ 4, abs(X,Z), goal2(X, Y).
```

The first answer to intermediate goal `Z ≥ 4, abs(X,Z)` adds constraint $Z \geq 4 \wedge X = Z$. Each of the possibilities for the `member` literal fails, and execution eventually searches the derivation corresponding to the second solution to the intermediate goal $Z \geq 4 \wedge X = -Z$ and finds the first solution to the entire goal in which $Y = 10$. In this case the size of the derivation tree explored has more than doubled.

9.6 CLP Languages with Dynamic Scheduling

We have only considered CLP systems which process the literals in a goal in a left to right manner. However, a fixed order of processing restricts the modes of usage for which a user-defined constraint can be used efficiently or correctly. To allow user-defined constraints to be used in multiple ways it would be useful to be able to "delay" evaluation of the constraint until its arguments were sufficiently constrained. That is to say, instead of using a literal processing order which is fixed

at compile time, we would like the system to employ a "dynamic" literal selection strategy which takes into account the current constraint store when choosing the literal.

For this reason, many recent CLP systems allow the programmer to annotate predicates with a "delay condition" which tells the system that the literals for the annotated predicate should not be evaluated until the current constraint store satisfies the delay condition.

For instance, recall the predicate from above which implements arithmetic disequality:

```
ne(X,Y)  :-  not(X=Y).
```

This predicate is only correct for the mode of usage in which both arguments are fixed when it is called. In a CLP system with delay, we can annotate the rules for `ne(X,Y)` with the condition that X and Y must be "ground" that is, take fixed values, before it can be evaluated:

```
:- delay_until(ground(X) and ground(Y), ne(X,Y)).
ne(X,Y) :- not(X=Y).
```

When evaluating goals involving the `ne` predicate, the CLP system will use a literal processing order which is basically left-to-right but which is safe in the sense that `ne(X,Y)` will not be rewritten until its delay condition is satisfied. Consider the goal

```
ne(X,Y), X=2, Y=3.
```

Using standard CLP evaluation it finitely fails which is incorrect. Using dynamic scheduling the goal has the single successful derivation (where the processed literal is underlined):

$$\langle ne(X,Y), \underline{X=2}, Y=3 \mid true\rangle$$
$$\Downarrow$$
$$\langle ne(X,Y), \underline{Y=3} \mid X=2\rangle$$
$$\Downarrow$$
$$\langle \underline{ne(X,Y)} \mid X=2 \wedge Y=3\rangle$$
$$\Downarrow$$
$$\langle \underline{not(X=Y)} \mid X=2 \wedge Y=3\rangle$$
$$\Downarrow$$
$$\langle \Box \mid X=2 \wedge Y=3\rangle.$$

Note that `ne(X,Y)` is not rewritten until the first state in which its delay condition holds.

We now examine the evaluation mechanism behind this example in more detail. In a CLP language with dynamic scheduling the programmer is able to provide declarations that delay the execution of a literal or goal until a particular delay

condition holds. A *delay condition*, *Cond*, simply takes a constraint and returns *true* or *false*, indicating if evaluation can proceed or should be delayed. Typical *primitive delay conditions* are `ground(X)`, which holds if `X` is constrained to take a fixed value, and `nonvar(X)`, which holds if `X` cannot take all possible domain values. More complex conditions include `Y` are identical `ask(c)` which holds if the constraint c is implied by the store and `exists a.X` $\leq$ `a` which holds if X is bounded above by some constant. Of course, as the preceding example shows, delay conditions may be constraint domain specific.

Primitive delay conditions can be combined to create more complex delay conditions. They can be conjoined, written ($Cond_1$ *and* $Cond_2$), or disjoined, written ($Cond_1$ *or* $Cond_2$).

Definition 9.3
A *delaying literal* is of the form *delay_until*$(Cond, G)$ where *Cond* is a delay condition and G is a goal.
Evaluation of G will be delayed until *Cond* holds for the current constraint store. We say that a constraint C *enables* the delaying literal *delay_until*$(Cond, Goal)$ if C satisfies *Cond*.

For instance, the constraint $X = 2 \wedge Y = 3$ enables

$$delay\_until(ground(X) \text{ and } ground(Y), ne(X, Y)),$$

while the constraint $X = Y$ does not. The enabling of a delay condition depends upon the solver. For example, the constraint $X \times X = 4 \wedge X \geq 0 \wedge Y = 1$ may not enable the above delaying literal if the solver is incomplete for nonlinears.

In CLP languages with dynamic scheduling there are two approaches for specifying the delay information:

- *Predicate-based*: In this case, the delaying literal appears as a declaration before the definition of the predicate and has the form

`:- delay_until(`*Cond*`,p(`$X_1, \ldots, X_n$`)).`

where $X_1, \ldots, X_n$ are distinct variables and *Cond* is a delay condition over a subset of those variables. Each user-defined constraint for the predicate p inherits the delay condition of the predicate.

- *Goal-based*: the delay condition is associated with a particular goal. In this case, the delaying literal `delay_until(`*Cond*`,`*Goal*`)` appears as a literal in the body of the goal or rule body and *Cond* is defined over the variables of that goal or rule body.

It is straightforward to use predicate-based declarations to imitate goal-based declarations, and vice versa. For example,

```
:- delay_until(ground(X) and ground(Y), ne(X,Y)).
ne(X,Y) :- not(X=Y).
```

and

```
ne(X,Y) :- delay_until(ground(X) and ground(Y), ne1(X,Y)).
ne1(X,Y) :- not(X=Y).
```

are equivalent although the first program uses predicate-based declarations while the second uses goal-based declarations. For simplicity, in this chapter we will only use predicate-based declarations.

We now describe how the standard evaluation mechanism given in Chapter 4 can be extended to handle delaying literals. Evaluation with dynamic scheduling means that literals are not necessarily processed in left-to-right order. We define a *literal selection strategy* to be a function that chooses which literal in a state is to be rewritten. We simply require that whenever a literal with a delay declaration is selected, the delay condition is enabled.

Definition 9.4
A literal selection strategy is *safe* if, whenever a literal L of the form $p(t_1, \ldots t_n)$ is selected in state $\langle G_0 \mid C_0 \rangle$ and there exists a delay declaration for p of the form $delay\_until(Cond, p(x_1, \ldots, x_n))$, then $C_0 \wedge x_1 = t_1 \wedge \cdots \wedge x_n = t_n$ enables $Cond$.
Note that enabling is solver dependent.
A (selection) derivation is *safe* if it employs a safe literal selection strategy.

Since the evaluation mechanism is required to use a safe literal selection strategy, it may happen that execution arrives at a state containing only delaying literals which are not enabled. We must generalize the definition of successful derivation and answer to cater for this case.

Definition 9.5
A state $\langle G \mid C \rangle$ is said to have *floundered* if G is a non-empty sequence of literals, none of whose delay conditions is enabled by C.
A safe derivation

$$\langle G_0 \mid C_0 \rangle \Rightarrow \cdots \Rightarrow \langle G_n \mid C_n \rangle$$

is *successful* if $solv(C_n) \not\equiv false$ and either $\langle G_n \mid C_n \rangle$ is floundered or G_n is the empty goal. The goal $G_n \wedge simplify(C_n, vars(\langle G_0 \mid C_0 \rangle) \cup vars(G_n))$ is a *qualified answer* to the state $\langle G_0 \mid C_0 \rangle$.

For example, the goal `ne(X,Y), X=2` has the single successful derivation

$$\begin{array}{c} \langle ne(X,Y), \underline{X = 2} \mid true \rangle \\ \Downarrow \\ \langle ne(X,Y) \mid X = 2 \rangle \end{array}$$

which gives the qualified answer $ne(X,Y) \wedge X = 2$.

We have already seen how we can use dynamic scheduling to robustly handle disequations. As another simple example, consider how we can improve the constraint

$$\langle \underline{not\_empty(L2)}, concat([L1, L2], L), L = [\,] \mid true \rangle$$
$$\Downarrow$$
$$\langle \underline{concat([L1, L2], L)}, L = [\,] \mid L2 = [\_|\_] \rangle$$
$$\Downarrow$$
$$\langle append(L1, T, L), \underline{concat([L2], T)}, L = [\,] \mid L2 = [\_|\_] \rangle$$
$$\Downarrow$$
$$\langle append(L1, T, L), \underline{L = [\,]} \mid L2 = [\_|\_] \wedge T = L2 \rangle$$
$$\Downarrow$$
$$\langle \underline{append(L1, T, L)} \mid L2 = [\_|\_] \wedge T = L2 \wedge L = [\,] \rangle$$
$$\Downarrow$$
$$\langle \blacksquare \mid false \rangle$$

Figure 9.4 Simplified derivation for concatenation with delay.

solver for sequences given in Section 9.1. We can modify the definition of append so that it delays until either the first or third argument is not a variable.

```
:- delay_until(nonvar(X) or nonvar(Z), append(X,Y,Z)).
append([], Y, Y).
append([A|X], Y, [A|Z]) :- append(X,Y,Z).
```

One safe (simplified) derivation for the goal

```
not_empty(L2), concat([L1,L2], L), L=[].
```

is illustrated in Figure 9.4. The derivation proceeds left-to-right until the third state. It is not safe to select the leftmost literal, $append(L1, T, L)$, in this state since both $L1$ and L are free. However, it is safe to select the next literal, $concat([L2], T)$. In the next state it is still not safe to select $append(L1, T, L)$ so the next literal $L = [\,]$ is selected. Now $append(L1, T, L)$ can be selected since L is no longer free. Rewriting with either rule immediately leads to failure. Therefore, with dynamic scheduling, the goal now behaves as desired and finitely fails.

One of the most important benefits of dynamic scheduling is that it facilitates the construction of simple, yet robust, constraint solvers in the CLP language itself. In particular, it makes it easy to write local propagation based solvers. For example, here is a simple local propagation solver for the Boolean constraints, negation (bnot) and conjunction (and), implemented using dynamic scheduling.

```
:- delay_until((ground(X) and ground(Y)) or
               (ground(X) and ground(Z)) or
               (ground(Y) and ground(Z)), and(X,Y,Z)).
and(0,0,0).
and(0,1,0).
and(1,0,0).
and(1,1,1).
```

```
:- delay_until(ground(X) or ground(Y), bnot(X,Y)).
bnot(0,1).
bnot(1,0).
```

The program works by delaying the selection of Boolean constraint literals until all but one argument is fixed. In this case the definitions are deterministic (except in two cases).

Compare execution of the program with and without delay declarations for the goal:

```
and(X, Y, Z), and(Y, Z, T), bnot(Y, 0), bnot(T, 0).
```

Using the delay declarations, every time a literal is selected there is only one rule which can succeed. First `bnot(Y,0)` is selected, which constrains Y to be 1, then `bnot(T,0)` is selected, which constrains T to be 1. Now the delay condition for `and(Y, Z, T)` is enabled. Rewriting this literal constrains Z to be 1. Finally, the literal `and(X, Y, Z)` is selected and rewriting constrains X to be 1. The simplified derivation tree contains 13 states and it is deterministic. Without delay declarations there are four answers to the first literal, and each of these answers leads to an answer to the second literal. Overall, the simplified derivation tree has 31 states.

Unfortunately, the and predicate is not deterministic for all modes of usage possible under the delay declarations. Consider the states $\langle and(X,Y,Z) \mid X = 0 \wedge Z = 0 \rangle$ and $\langle and(X,Y,Z) \mid Y = 0 \wedge Z = 0 \rangle$. For each state there are two possibilities for the remaining variable X or Y. But either possibility satisfies the constraint so we can simply ignore the constraint in these circumstances. Therefore we can rewrite the definition of and so as to make it deterministic as follows:

```
:- delay_until((ground(X) and ground(Y)) or
               (ground(X) and ground(Z)) or
               (ground(Y) and ground(Z)), and(X,Y,Z)).
and(X,Y,Z) :-
    (ground(Z), Z = 0 ->
        (ground(X), X = 0 -> true
        ; (ground(Y), Y = 0 -> true
        ; and2(X,Y,Z)
        ))
    ; and2(X,Y,Z)
    ).

and2(0,0,0).
and2(0,1,0).
and2(1,0,0).
and2(1,1,1).
```

If Z is fixed and zero and either X is fixed and zero, or Y is fixed and zero, then the constraint holds and nothing more need be done. In the other cases we just use the previous definition. The definition makes use of the built-in ground as part of the tests in the if-then-else literals. This tests to see if its argument has a fixed value. Since it is not part of a delay condition, evaluation does not delay if the argument is not fixed, instead it fails.

We can see from these examples that dynamic scheduling allows us to write user-defined constraints which are more "robust" and which behave as if they are being evaluated by a well-behaved solver. Indeed, barring non-termination, standard evaluation of user-defined constraints is complete—either the system finds a solution or else it finitely fails. Dynamic scheduling allows the evaluation mechanism to return an "unknown" answer indicating that the evaluation mechanism will not further evaluate a goal it has created because the delay declarations indicate that this will be too expensive or may not terminate

In starred Section 4.8, we saw that the answers to a goal are independent of the literal selection strategy. It is this key property which makes programming with dynamic scheduling feasible, since the programmer knows that, regardless of the precise order of selection chosen with dynamic scheduling, evaluation will still give the same answers. Fortunately, independence from the literal selection strategy continues to hold for dynamic scheduling as long as the delay conditions are reasonably behaved. The key requirement is that the delay conditions must be *downwards closed*, that is once the delay condition is satisfied, it will continue to hold for the remainder of the forward computation. More exactly, delay condition $Cond$ is downwards closed if for any two constraints C_1 and C_2 such that $C_2 \rightarrow C_1$, whenever $Cond$ holds for C_1, it also holds for C_2. Other requirements are that the delay condition should not take variable names into account and that its behaviour cannot depend on variables which are not mentioned in the condition.

Delay conditions in most CLP languages with dynamic scheduling satisfy these properties. This allows the underlying implementation to use a fixed literal selection strategy, for example, always selecting the leftmost literal which is enabled.

Since Chapter 4, we have introduced constructs, such as once and if-then-else that can destroy the independence from literal processing order. When we make use of them with dynamic scheduling, we need to take care that they are used in a manner in which the answer does not depend on the order in which literals are evaluated. For example, in the deterministic definition for and above, the tests in the if-then-else's either succeed or fail, for all possible modes of usage under which the literal can be selected, but never constrain a non-fixed variable. Hence they are independent of the order in which literals are processed.

9.7 (*) Meta Programming

One interesting application of higher-order programming is for meta-programming. *Meta-programs* are programs which manipulate other programs as data. For example, they may analyse, compile, transform or execute other programs. Because programs can easily be represented by tree data structures, CLP languages are well suited for writing meta-programs.

One simple example of a meta-program is an interpreter for $CLP(Tree)$. This is a program that, given a program and goal, imitates the execution of the $CLP(Tree)$ system. This may not sound very useful, but once we have written this program we can modify it to change the standard CLP evaluation mechanism.

The first consideration when writing a meta-program is how to represent the program which will be manipulated by the meta-program. Like goals and literals, CLP programs may be represented by terms and manipulated by means of tree constraints. For instance, the rule

```
traverse(node(T1,I,T2), L) :-
     traverse(T1, L1),
     traverse(T2, L2),
     append(L1, [I|L2], L).
```

can be represented by the term

```
:-(traverse(node(T1,I,T2), L),
     ','(traverse(T1, L1),
     ','(traverse(T2, L2),
     append(L1, [I|L2], L))))
```

illustrated in Figure 9.5. Recall that $[I|L2]$ is notation for $.(I, L2)$.

Representing goals using the comma "," notation is ugly and it also causes difficulty when we wish to build and decompose goals. Instead we will represent the body of a rule as a list of literals, and represent a rule by a two argument predicate `lrule` whose first argument is the head and whose second argument is the body. For example the `traverse` rule above is represented by the fact

```
lrule(traverse(node(T1,I,T2),L),
      [traverse(T1,L1),
       traverse(T2,L2),
       append(L1,[I|L2],L)]).
```

Our meta-interpreter, `derive`, takes a goal, *Goal*, as its sole argument. It selects a literal *Lit* from *Goal*, breaking the rest of the goal into the parts before, *Pre*, and after, *Post*, the selected literal. The replacement, *Rep*, of the literal in the goal part is then inserted between *Pre* and *Post* to form the new goal, that is to say, $NewGoal = Pre :: Rep :: Post$.

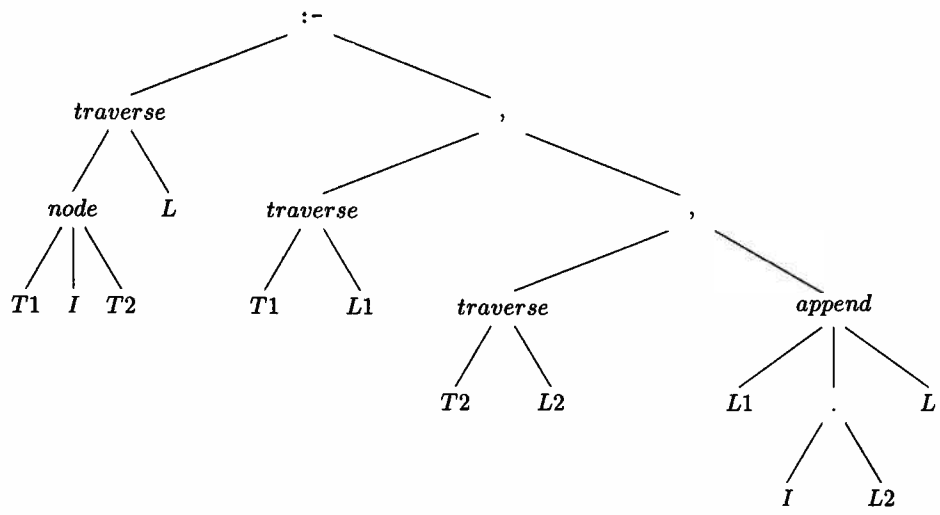

Figure 9.5 A term describing the rule for `traverse`.

If the literal is a tree constraint, the interpreter simply uses `call` to add the equation to the store and the replacement is the empty list. Otherwise `lrule` is used to replace a user-defined constraint by the body of a matching rule. The renaming of the rule and the constraints equating the selected literal and the head of the rule are handled implicitly by the underlying CLP system.

```
derive([]).
derive(Goal) :-
     select(Goal, Lit, Pre, Post),
     replace(Lit, Rep),
     append(Pre, Rep, Tmp),
     append(Tmp, Post, NewGoal),
     derive(NewGoal).

replace(Lit, Rep) :-
     (constraint(Lit) ->
          call(Lit), Rep = []
     ;
          lrule(Lit, Rep)
     ).

constraint(_ = _).
```

The standard evaluation mechanism is simply to take the first literal in the goal

```
select([Lit|Rest], Lit, [], Rest).
```

Consider the append program. It is represented by

```
lrule(append([], Y, Y), []).
lrule(append([A|X], Y, [A|Z]), [append(X, Y, Z)]).
```

The simplified successful derivation for the goal

```
derive([append([a,b], [c,d], T)]).
```

is as follows, where we only show the states in which `derive` literals are selected.

$$\langle derive([append([a,b],[c,d],T)]) \mid true\rangle$$
$$\Downarrow_*$$
$$\langle derive([append([b],[c,d],T1)]) \mid T = [a|T1]\rangle$$
$$\Downarrow_*$$
$$\langle derive([append([\,],[c,d],T2)]) \mid T = [a,b|T2]\rangle$$
$$\Downarrow_*$$
$$\langle derive([\,]) \mid T = [a,b,c,d]\rangle$$
$$\Downarrow_*$$
$$\langle \Box \mid T = [a,b,c,d]\rangle$$

Now that we have written a meta-interpreter we can modify it to, for example, select literals in a different way. Consider the constraint solver for sequences defined in Section 9.1. The goal

```
not_empty(L2), concat([L1,L2],L), L=[].
```

ran forever because there are an infinite number of answers to the first `append` literal that appears in the derivation. We have seen how, using dynamic scheduling, we can control the derivation so that the goal will determine an answer. We now modify our meta-interpreter to mimic dynamic scheduling.

There are only a finite number of answers to any call to `append` if either the first or third argument has fixed length. Since the left-to-right selection rule is explicitly programmed into the meta-interpreter, we can modify the meta-interpreter so that it only selects `append` literals when either the first or third argument is not a variable. We need only change `select` to:

```
select([Lit0 | Rest], Lit, Pre, Post) :-
     (Lit0 = append(X,Y,Z) ->
          (one_nonvar(X,Z) ->
               Lit = append(X,Y,Z), Pre = [], Post = Rest
          ;
               Pre = [Lit0 | Pre1],
               select(Rest, Lit, Pre1, Post)
          )
     ;
          Lit = Lit0, Pre = [], Post = Rest ).
```

```
one_nonvar(X,_) :- nonvar(X).
one_nonvar(_,Z) :- nonvar(Z).
```

The `one_nonvar(X,Z)` literal checks whether one of X and Z is not a variable. It makes use of the library function `nonvar` which succeeds unless its argument is a variable that can take any value.

An example (simplified) derivation of the goal

```
derive([not_empty(L2), concat([L1,L2],L), L=[]]).
```

is shown below, where the literal selected by the meta-interpreter is underlined.

$$\langle derive([\underline{not\_empty(L2)}, concat([L1, L2], L), L = []]) \mid true\rangle$$
$$\Downarrow_*$$
$$\langle derive([\underline{concat([L1, L2], L)}, L = []]) \mid L2 = [\_|\_]\rangle$$
$$\Downarrow_*$$
$$\langle derive([append(L1, T, L), \underline{concat([L2], T)}, L = []]) \mid L2 = [\_|\_]\rangle$$
$$\Downarrow_*$$
$$\langle derive([append(L1, T, L), \underline{L = []}]) \mid L2 = [\_|\_] \wedge T = [\_|\_]\rangle$$
$$\Downarrow_*$$
$$\langle derive([\underline{append(L1, T, L)}]) \mid L2 = [\_|\_] \wedge \ \wedge T = [\_|\_] \wedge L = []\rangle$$
$$\Downarrow_*$$
$$\langle \blacksquare \mid false\rangle$$

In the third and fourth states the first `append` literal cannot be selected since both $L1$ and L can take any value. Thus, the `concat` literal is selected in the third state and the $L = []$ constraint in the fourth state. Finally, the `append` literal can be selected since L is now constrained to be the empty list. Rewriting with either of the rules in `append` leads to failure since T is a non-empty list. Therefore, the system has determined that there is no answer to the goal.

The meta-interpreter described above, in effect, implements a dynamic scheduling literal selection rule. Indeed the above derivation corresponds to that shown in Figure 9.4 which was obtained by using dynamic scheduling. One advantage of meta-programming over dynamic scheduling is that we can program arbitrary selection rules using the CLP language, rather than relying only on the delay conditions provided by the CLP system. Meta-interpretation, however, is much more expensive than using built-in dynamic scheduling facilities. Generally, if dynamic scheduling facilities are available, these should be used in preference to meta-programming.

9.8 (*) Library Predicates

Throughout this book we have seen numerous examples of library predicates, often called *built-ins*, provided by our fictional CLP system. Unfortunately, there is, as yet, no agreed upon standard library of built-ins. Each system provides different built-in predicates. Indeed, since many of the built-ins are constraint domain specific, it is not possible for every system to provide all of them.

There is, however, rough consensus on the functionality that should be provided by many of the built-ins. Broadly speaking, the most useful built-ins can be grouped into the following six classes: program consultation and inspection; input and output; debugging; examination of the constraints on a variable and its type; database programming; and higher-order predicates. Built-ins in these classes are largely constraint domain independent and, although the syntax may differ, most CLP systems provide essentially the same functionality.

In the practical exercises we have already seen how to consult, reconsult and list a CLP program. We have also briefly discussed the most common higher-order built-ins in Section 9.4. In this section, we briefly describe the built-ins in each of the other classes. We base our discussion loosely on the built-ins of SICStus Prolog and $CLP(\mathcal{R})$, but other systems provide corresponding library predicates. At least a cursory knowledge of these built-ins is necessary for more advanced programming with CLP.

Unlike user-defined constraints and primitive constraints, understanding the behaviour of most of these built-ins relies on knowledge of the order of literal evaluation and the content of the constraint store at the time the built-in is evaluated. For this reason, programming with built-ins is often quite subtle and may lead to unexpected errors—we advise the programmer to use them carefully and sparingly.

9.8.1 Input and Output

Any practical programming language must provide facilities for input and output. In CLP systems the built-in `read` is used to read the next expression from the input stream. Usually the input stream is the sequence of characters typed on the key board, but the user may also direct the input-stream to be the characters in a file. Evaluation of the literal `read(t)` essentially adds the constraint $t = t'$ to the current goal where t' is the next expression on the input stream. A full-stop (or period) "." is used to terminate an input expression.

For example, evaluation of the $CLP(\mathcal{R})$ goal

```
read(X), read(Y).
```

during which the user types

```
1.01.  2.
```

gives the single answer $X = 1.01 \wedge Y = 2$. Some CLP systems restrict the argument to `read` to be a variable.

If the input stream is finished, that is if the input file has been fully read, a special expression indicating end of file is returned. The value of this expression is system dependent.

The built-in `write` is used to write the value of an expression to the output stream. Usually the output stream is to a terminal, but the user may also direct output to a file. Evaluation of the literal `write(t)` writes the value of t as implied by the current constraint to the output stream. Generally, if t has no fixed value then a representation of a variable is printed, although for tree constraints the solved form of t will be printed.

Evaluation of a `write` literal does not affect the constraint store. The predicate `write` can also be called with a string as an argument. In this case the string will be written to standard output. The built-in `nl` writes the new-line character to standard output.

For example, evaluation of the $CLP(\mathcal{R})$ goal

```
X=2, Y=3, write("X + Y is: "), write(X+Y), nl.
```

outputs the following

```
X + Y is: 5
```

Evaluation of

```
X=2, write("X + Y is: "), write(X+Y), nl.
```

will output the following

```
X + Y is: _t3
```

The `_t3` indicates an arithmetic variable.

The `write` command, although provided in most CLP systems is an historical relic, and does not really fit with the constraint paradigm. Some systems provide a facility for writing the constraints affecting a set of chosen variables. For example the `dump` facility in $CLP(\mathcal{R})$ does this. The literal `dump(`$[V_1, \ldots, V_n]$`)` writes the constraints in $simpl(\{V_1, \ldots, V_n\}, C)$ where C is the current constraint store. The built-in `dump` allows interactive programs to provide information about variables of interest.

For example, evaluation of the goal

```
X < 2, Z=X+Y, Y < 3, dump([Z]).
```

writes the following

```
Z < 5
```

If the literal `dump` is replaced by `write(Z)` the output of the goal is merely `_t3`.

As a more detailed example of the use of these built-ins, consider the `deriv` program from Section 9.2. The following goal when executed prompts the user to enter an arithmetic function f in terms of variable x and a value for x and will write out the value of f and the slope of the function, for this value of x.

```
write("Input function: "), read(F),
write("Input x value: "), read(X),
deriv(F,DF), evaln(F,X,FX), evaln(DF,X,DFX),
write("The value of function "), write(F), nl,
write("when x = "), write(X), write(" is "), write(FX), nl,
write("and the slope is "), write(DFX), nl.
```

If the goal above is executed, the following session occurs (where user input is given in *emphasized text*)

```
Input function: mult(sine(x),plus(x,2)).
Input x value: 1.
The value of function mult(sine(x), plus(x, 2))
when x = 1 is 2.52441
and the slope is 2.46238
```

9.8.2 Variable Examination

Apart from built-ins for input/output, typical CLP systems provide a number of built-ins so that the programmer can determine the "type" of a variable, that is what kind of constraints it has been involved in, and how "constrained" the variable is. Typical built-ins are:

`atom(X)`: This succeeds if the current constraint store contains tree constraints constraining X to be a tree which is a constant.

`compound(X)`: This succeeds if the current constraint store contains tree constraints constraining X to be a tree which is not a constant.

`arithmetic(X)`: This succeeds if the current constraint store contains arithmetic constraints which the solver can determine constrain X to take a fixed arithmetic value.

`integer(X)`: This succeeds if the current constraint store contains arithmetic constraints which the solver can determine constrain X to take a fixed integer value.

`ground(X)`: This succeeds if the constraint solver can determine that the current store implies that X has a fixed value.

`nonvar(X)`: This succeeds if the constraint solver can determine that the current constraint store implies that X is not free. That is to say, it cannot take all values in the constraint domain. Unfortunately, this behaviour is only guaranteed when the variable has only been involved in tree constraints or the solver can determine that X must take a fixed value.

`var(X)`: This succeeds if the constraint solver can determine that X is free, that is, it can take all values in the constraint domain. Again, this behaviour is only guaranteed when the variable has only been involved in tree constraints. It is guaranteed to fail if the solver can determine that X has a fixed value.

We have previously seen the built-in `arithmetic` in Section 9.2 and the built-in `ground` in Section 9.6. The built-in `ground(X)`, as opposed to the delay condition `ground(X)`, does not *delay* until X is fixed, it just performs a check, failing if the argument is not fixed. The built-in `nonvar(X)` is similar. Other built-ins described above act analogously to `arithmetic(X)`.

For instance, the goal

```
X = a, atom(X).
```

will succeed while the goals

```
X = f(Y,a), atom(X).
```

and

```
X > Y, atom(X).
```

will fail. The goal

```
X = f(Y,b), ground(X), Y = a.
```

will fail, while

```
X = f(Y,b), Y = a, ground(X).
```

will succeed. The goal

```
X = f(a,Y), var(X).
```

will fail, the goal

```
X = Y, var(X).
```

will succeed as will the goal

```
X = f(Y), nonvar(X).
```

For CLP systems that support finite domain constraints, there are a number of built-ins for accessing the current domain of a variable. We have already met these in Chapter 8.

`mindomain(X,M)`: Sets M to be the minimum value of the domain of the variable X in the current constraint store. At the time of evaluation, M should be free.

`maxdomain(X,M)`: Sets M to be the maximum value of the domain of the variable X in the current constraint store. Again, at the time of evaluation, M should be free.

`dom(X,L)`: Sets L to the list of all values in the domain of the variable X in the current constraint store. At the time of evaluation, L should be free.

For example, the goal

```
X :: [0..8], X ≥ 4, mindomain(X,Min), maxdomain(X, Max).
```

constrains Min to be 4 and Max to be 8. The goal

```
X :: [0..8], X ≥ 3, X ≠ 5, dom(X,L).
```

constrains L to be $[3,4,5,6,7,8]$ if disequations are handled by bounds consistency, or $[3,4,6,7,8]$ if disequations are handled by arc consistency.

There are also three useful built-ins for manipulating terms: `functor`, `arg` and `=...` These allow the programmer to build or access terms when the arity and name of the tree constructor is unknown.

The literal `functor(T,F,N)` succeeds if T is a term constructed from the tree constructor F with N children. At the time of evaluation either T must be constrained to be a non-variable term or F and N must be fixed, otherwise a runtime error occurs. For instance, the goal

```
X = record(3,Y), functor(X,F,N).
```

succeeds with answer

$$X = record(3,Y) \wedge F = record \wedge N = 2,$$

while the goal

```
F = record, N = 2, functor(X,F,N).
```

succeeds with answer

$$X = record(\_Y,\_Z) \wedge F = record \wedge N = 2.$$

The literal `arg(N,T,A)` succeeds if the term T has term A as its N^{th} argument. At the time of evaluation T must be constrained to be a non-variable term and N must be fixed, otherwise a runtime error occurs. For instance, the goal

```
X = record(3,Y), arg(1,X,A).
```

succeeds with answer $X = record(3,Y) \wedge A = 3$, while `arg(N,record(3,Y),A)` gives a runtime error.

The literal `T =.. L` provides similar functionality to `functor` and `arg`.[1] At the time of evaluation either T must be constrained to be a non-variable term or L must be constrained to be a fixed length list whose first argument is a tree constructor. If T is constrained to be a term, then L is constrained to be a list $[F|As]$ where

1. Yes, it really is "=.." which is pronounced "univ".

F is a constant with the same name as the tree constructor and As is a list of the children of T. If L is a fixed length list with first element constant F and remainder As then T is constrained to be a term with tree constructor F and children As.

We can use `functor` and `arg` to give an alternate definition of vectors in CLP languages.

```
init(N, Vec) :- functor(Vec, vec, N).
access(Vec, I, E) :- arg(I, Vec, E).
```

A vector $\langle v_1, \ldots v_n \rangle$ with n elements is represented by a tree with n children. The literal `init(N, Vec)` initializes Vec to be a term with N children each of which is a distinct variable. The literal `access(Vec, I, E)` equates E with the I^{th} element v_I of the vector Vec. The advantage of this representation over the list representation used earlier is that access to a particular element in the vector takes only constant time.

The following program for predicate `variables` illustrates many of these built-ins. The predicate `variables(T,L)` succeeds if L is the list of variables occurring in term T. The program makes use of an accumulator. The auxiliary predicate `variables_acc(T,L1,L2)` succeeds if list $L2$ is the result of appending the variables in T to the list of variables $L1$. The list $L1$ acts as an accumulator. The auxiliary predicate has three rules in its definition. The first rule is for the case when T is a variable. It is simply concatenated to $L1$ to give the new list $L2$. The second rule is for when T is a constant. In this case $L1$ simply equals $L2$. The final case is when T is a more complex term with arguments. The predicate `variables_in_args` iterates through the arguments in T appending the variables in each to the list by recursively calling `variables_acc`.

```
variables(T,L) :- variables_acc(T,[],L).

variables_acc(X, L, [X|L]) :- var(X).
variables_acc(X, L, L) :- atom(X).
variables_acc(T, L, L1) :-
     compound(T),
     functor(T,_,A),
     variables_in_args(T, A, L ,L1).

variables_in_args(_T, 0, L, L).
variables_in_args(T, A, L, L2) :-
     A > 0,
     arg(A, T, TArg),
     variables_acc(TArg, L, L1),
     variables_in_args(T, A-1, L1, L2).
```

As an example, the goal `variables(f(X,g(a,Y,X)),L)` returns the answer $L = [X, Y, X]$. It is important to realise that this predicate computes the variables in a term in the context of the current constraint store. Thus, the goal

```
T = harald, variables(T,L).
```

returns the answer $L = [\,]$, not $[T]$ as the reader may have expected.

As another example, we can use the term manipulation built-ins together with dynamic scheduling to build a more complete constraint solver for disequations if the system does not provide this capability.

```
ne(X,Y) :- neq(X,Y,V,true,false).

:- delay_until((nonvar(X) and nonvar(Y)) or ground(V),neq(X,Y,V,_,_)).
neq(X,Y,V,Pr,Nx) :-
     (ground(V) -> true
     ;    functor(X, F, N), functor(Y, G, M),
          (F = G, N = M ->
               X =.. [_|Xs], Y =.. [_|Ys],
               neqs(Xs, Ys, V, Pr, Nx)
          ;    V = true )
     ).

neqs([], [] , V, Pr, Pr).
neqs([X|LX],[Y|LY],V, Pr, Nx) :-
     neq(X,Y,V,Pr,In),
     neqs(LX,LY,V,In,Nx).
```

The literal `ne` does not wait until its arguments are fully ground. Instead it recursively processes the corresponding subterms of its arguments, determining if the current store implies the subterms must be unequal or, if it cannot do this, sets up delayed calls on subterms that might be found to be unequal in the future. Evaluation leads to calls of the form `neq(X,Y,V,Pr,Nx)` where X and Y are corresponding subterms. Such a call sets V to *true* if X and Y cannot be equal, and sets Pr equal to Nx if X and Y must be equal. The literal delays until both X and Y are not variables or V is fixed. V is a communication variable shared between all of these calls to `neq`. If it becomes ground, it causes all the delayed literals to wake up and succeed. V is set to *true* whenever a literal `neq(X,Y,V,Pr,Nx)` is found in which X and Y must be different. The remaining arguments Pr and Nx chain these `neq` literals together. When X and Y are both non-variables then the literal `neq(X,Y,V,Pr,Nx)` is replaced by `neq` literals among their arguments which chain the Pr and Nx variables. Whenever X and Y are found to be identical, Pr and Nx are equated. If all of the X and Y arguments in the `neq` literals are eventually found to be equal, the two values *true* and *false* given as the initial chain values are equated and failure results.

Consider evaluation of the goal

```
ne(X, Y), X = f(a, U), Y = f(T, b), U = b, T = a.
```

The simplified derivation (with some states omitted) has the form illustrated in Figure 9.6 where the rewritten literal(s) is underlined.

$$\langle \underline{ne(X,Y)}, X = f(a,U), Y = f(T,b), U = b, T = a \mid true \rangle$$
$$\Downarrow$$
$$\langle neq(X,Y,V,true,false), \underline{X = f(a,U), Y = f(T,b)}, U = b, T = a \mid true \rangle$$
$$\Downarrow_*$$
$$\langle \underline{neq(f(a,U), f(T,b), V, true, false)}, U = b, T = a \mid X = f(a,U) \wedge Y = f(T,b) \rangle$$
$$\Downarrow_*$$
$$\langle \underline{neqs([a,U],[T,b],V,true,false)}, U = b, T = a \mid X = f(a,U) \wedge Y = f(T,b) \rangle$$
$$\Downarrow_*$$
$$\langle neq(a,T,V,true,In), neq(U,b,V,In,false), \underline{U = b}, T = a \mid X = f(a,U) \wedge Y = f(T,b) \rangle$$
$$\Downarrow$$
$$\langle neq(a,T,V,true,In), \underline{neq(b,b,V,In,false)}, T = a \mid X = f(a,b) \wedge Y = f(T,b) \wedge U = b \rangle$$
$$\Downarrow_*$$
$$\langle neq(a,T,V,true,In), \underline{neqs([],[],V,In,false)}, T = a \mid X = f(a,b) \wedge Y = f(T,b) \wedge U = b \rangle$$
$$\Downarrow_*$$
$$\langle neq(a,T,V,true,false), \underline{T = a} \mid X = f(a,b) \wedge Y = f(T,b) \wedge U = b \rangle$$
$$\Downarrow$$
$$\langle \underline{neq(a,a,V,true,false)} \mid X = f(a,b) \wedge Y = f(a,b) \wedge U = b \wedge T = a \rangle$$
$$\Downarrow_*$$
$$\langle \underline{neqs([],[],V,true,false)} \mid X = f(a,b) \wedge Y = f(a,b) \wedge U = b \wedge T = a \rangle$$
$$\Downarrow_*$$
$$\langle \blacksquare \mid false \rangle$$

Figure 9.6 Simplified derivation for more complete `ne`.

In the third state, when X and Y are both non-variable, the `neq` literal is selected. Since they have the same tree constructor and number of children, the corresponding children are constrained by `neq` literals, in the chain

```
neq(a,T,V,true,In), neq(U,b,V,In,false).
```

The intermediate variable *In* allows one of these `neq` literals to equate the last two arguments, but not both. In the sixth state, the second `neq` literal is selected. Since the two terms *b* and *b* are identical and have no arguments, the call to `neqs([],[],V,In,false)` sets *In* to *false*. In the ninth state the remaining `neq` literal is selected. Since the tree constructors are identical and have no children, the call to `neqs([],[],V,true,false)` attempts to equate *true* and *false* and fails. Since all corresponding subterms of the initial two variables X and Y are shown to be identical the disequation is false.

As another example, consider evaluation of the goal

```
ne(X, Y), X = f(a, U), Y = f(T, b), U = a.
```

The simplified derivation proceeds similarly to that above until it reaches the following state:

$$\langle neq(a,T,V,true,In), neq(U,b,V,In,false), \underline{U=a} \mid X = f(a,U) \wedge Y = f(T,b) \rangle$$
$$\Downarrow$$
$$\langle neq(a,T,V,true,In), \underline{neq(a,b,V,In,false)} \mid X = f(a,a) \wedge Y = f(T,b) \wedge U = a \rangle$$
$$\Downarrow_*$$
$$\langle neq(a,T,V,true,In), \underline{V=true} \mid X = f(a,a) \wedge Y = f(T,b) \wedge U = a \rangle$$
$$\Downarrow$$
$$\langle \underline{neq(a,T,true,true,In)} \mid X = f(a,a) \wedge Y = f(T,b) \wedge U = a \rangle$$
$$\Downarrow_*$$
$$\langle \Box \mid X = f(a,a) \wedge Y = f(T,b) \wedge U = a \rangle$$

In the second state the `neq` literal is replaced by $V = true$ since the tree constructors of the first two arguments are not the same. Therefore the original terms cannot be identical. When V is set to true, the remaining `neq` literal is selected and replaced with *true*. The goal succeeds since X is constrained to be $f(a,a)$ and Y is constrained to be $f(T,b)$ which cannot be equal.

9.8.3 Database Built-ins

Most CLP languages provide built-ins which allow the programmer to add, examine or remove the rules in a program while the program is executing. Needless to say, the use of such built-ins is fraught with peril since they allow the programmer to write complex self-modifying code. CLP programmers rarely use these built-ins—they are primarily intended for implementation of the other built-ins.

The three main database built-ins are:

`clause`: This returns the rules in a program.

`assert`: This adds a rule to a program.

`retract`: This removes a rule from a program.

The literal `clause(H, B)` succeeds if the rule H `:-` B occurs in the program. At the time of the call the head, H, must be constrained to be a non-variable term. A fact is represented by a rule with the body *true*. A call to `clause` will backtrack through all rules matching the arguments.

The literal `assert(H)` adds the fact H while `assert((H :- B))` adds the rule H `:-` B to the program. The extra parentheses in the second call are necessary because of the precedence of the "`:-`" operator. At the time of the call, the head H, and body B, if present, must be constrained to have the syntax of a user-defined constraint and a goal respectively. Upon backtracking, a call to `assert` will fail, leaving the rule in the program.

The final database built-in is `retract`. The call `retract(H)` removes the fact H while `retract((H :- B))` removes a rule H `:-` B from the program. As for `assert`, at the time of the call H and B, if present, must be constrained to have the syntax of a user-defined constraint and a goal respectively. Upon backtracking a call to `retract` will remove another fact or rule matching the arguments. Those rules previously removed from the program by the call remain removed. If no rules match the arguments then it will fail.

In general, good CLP programming style is to use `assert` and `retract` only to add or delete facts from the program. Furthermore, at the time they are asserted, all variables in the facts should have fixed values. Adding or deleting complex rules containing variables is error-prone and best left to the CLP system implementor. Adding or deleting rules with arithmetically constrained variables (that are not fixed by the solver) is not fully supported by many CLP systems.

One use of database predicates is to define global variables, that is variables that can be accessed from any part of the program. Imagine we wish to profile the use of `member` within the intersection program from Example 7.1. Recall the definition of `intersect`. It succeeds if its two arguments are lists which have a common element.

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

member(X, [X|_]).
member(X, [_|R]) :- member(X, R).
```

We wish to add a flag to `member` which will turn profiling on or off. If the flag is set to 1, we wish to print the solved form of the arguments whenever a call to `member` is evaluated and whenever an answer is found to the call. If the flag is not set to 1, evaluation of `member` should work silently.

We could add an extra argument to `member` and `intersect` and pass the flag's value in to the call through this argument. This has the disadvantage that we also have to modify `intersect`. Instead we can use the database to store the flag value v, using a fact of the form `flag(`v`)`. This can be accessed in `member` as follows and does not require us to modify the code for `intersect`.

```
member(X,Y) :-
     (clause(flag(1),true) ->
          write("Call: "), write(member(X,Y)), nl
     ; true),
     member1(X,Y),
     (clause(flag(1),true) ->
          write("Exit: "), write(member(X,Y)), nl
     ; true).

member1(X, [X|_]).
member1(X, [_|R]) :- member(X, R).
```

The simplified derivation tree for the goal `intersect([1,2], [0,2,4])` is illustrated in Figure 9.7.

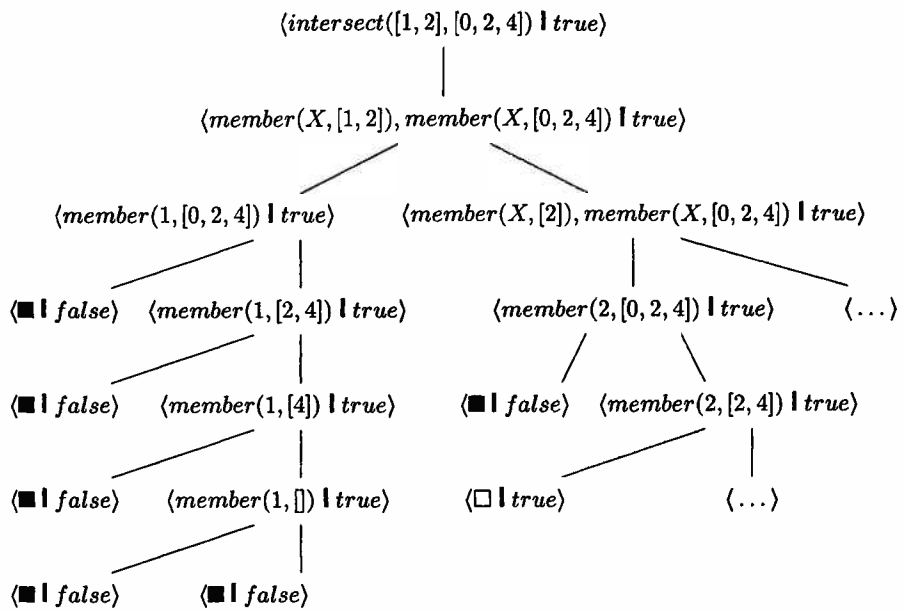

Figure 9.7 Simplified derivation tree for `intersect`.

Evaluation of the goal

```
assert(flag(1)), intersect([1,2], [0,2,4]).
```

Results in the answer *true* and the output

```
Call: member(X, [1, 2])
Exit: member(1, [1, 2])
Call: member(1, [0, 2, 4])
Call: member(1, [2, 4])
Call: member(1, [4])
Call: member(1, [])
Call: member(X, [2])
Exit: member(2, [2])
Exit: member(2, [1, 2])
Call: member(2, [0, 2, 4])
Call: member(2, [2, 4])
Exit: member(2, [2, 4])
Exit: member(2, [0, 2, 4])
```

A call to `member` occurs in the derivation tree whenever the first literal in the state has the predicate name `member`. An exits from that call, indicating that an answer has been found, occurs just before the next literal, in the state in which the original call occurs, is rewritten. For example, the first exit corresponds to the first answer to the first `member` literal.

The goal

```
assert(flag(0)), intersect([1,2], [0,2,4]).
```

simply returns the answer *true* and does not print anything.

For the intersection program, it may seem easier to add the extra argument. However, for larger programs using a global flag in the database is more convenient since it means we only have to modify the definition of `member` rather than that of all the predicates which indirectly call `member`.

Another use of database predicates is for communicating information from one branch in a derivation tree to another. For example, we can define facilities for counting using the database predicates as follows:

```
init :- assert(counter(0)).
inc :- once(retract(counter(X)), X1 = X + 1, assert(counter(X1))).
result(X) :- retract(counter(X)).
```

The predicate `init` initializes the counter to value 0, `inc` increments the counter, and `result(X)` removes the counter and returns the value in X. The `once` in the definition of `inc` is important since otherwise, if a literal after `inc` fails, backtracking will cause the `retract` to remove the already incremented counter value, and assert a new counter value of one more.

Suppose we wish to count the number of answers to a goal. First we shall illustrate this using a very simple goal `X` $\geq$ `2, member(X, [0,3,7])`. We need to initialize the counter, execute the goal and, whenever we find an answer, increment the counter. To force the execution to find all answers we fail and look for another answer. After all answers have been found, the value in the counter records the number of answers found. The following program encodes this:

```
simple_count(_) :- init, X ≥ 2, member(X, [0,3,7]), inc, fail.
simple_count(N) :- result(N).
```

Note the use of the built-in `fail` to force the body of the first rule to fail. A simplified derivation tree for the goal `simple_count(N)` is shown in Figure 9.8. Derivation steps evaluating the predicates `init`, `inc` and `result` are omitted, but the resulting changes to the database are marked alongside of the derivation step. The annotation $+fact$ indicates that $fact$ is asserted into the database, while $-fact$ indicates that $fact$ is retracted.

We can write a higher-order predicate that counts the number of answers to an arbitrary goal as follows:

```
count_answers(G,_) :- init, call(G), inc, fail.
count_answers(_,N) :- result(N).
```

The goal `count_answers(G,N)` returns in N the number of answers to the goal G.

Although the (self-imposed) restriction that `assert` is only used in a mode in which all of the variables in it are fixed means that arbitrary constraints cannot be communicated across different branches in a derivation tree, it still allows the CLP

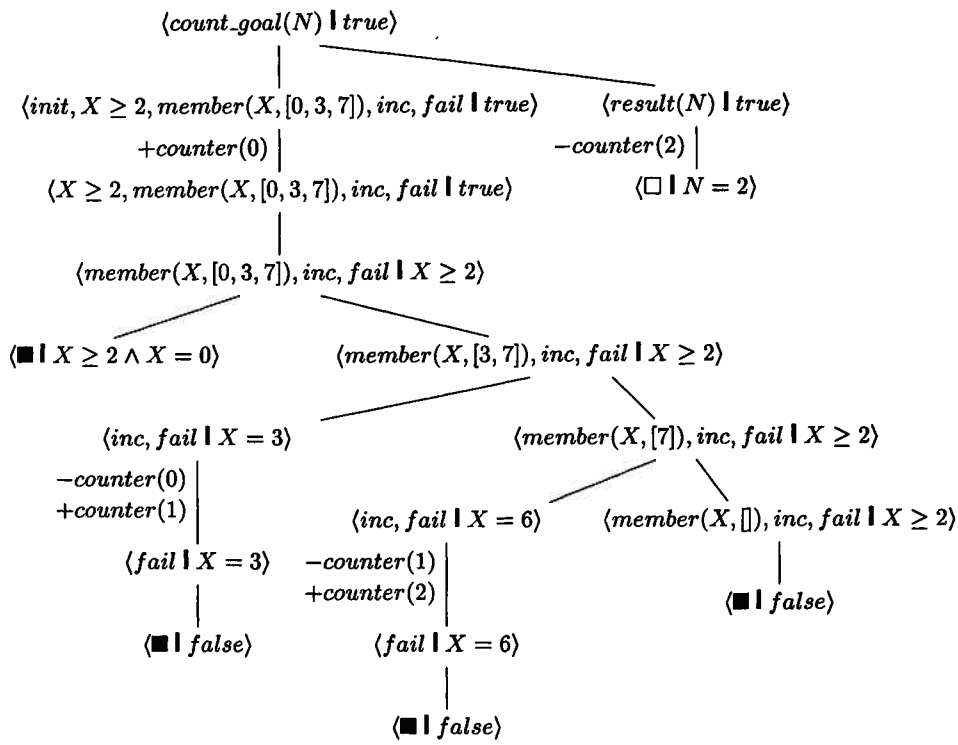

Figure 9.8 Using database predicates to count solutions.

programmer to modify the default search strategy and to program built-ins such as `minimize` for goals.

As a more complex example, consider how one might implement the higher-order built-in `minimize(G,E)`. The first part of the evaluation is to find the minimum value M of the expression E to be minimized. Then G is called to find any answer in which E has this value. The most difficult part is to determine the minimum value, since we do not want to look at all answers but, rather, wish to use the current bound to prune those answers to G which cannot tighten this bound. The following program does this by repeatedly solving the goal G after the bound on E has been tightened. Thus it is very similar to retry_int_opt defined in Section 3.6. Communication of the new bound between different calls to G is by means of the predicate `apply_new_bound(E)` which removes the current bound on B from the database and constrains $E < B$ so as to find a better bound, and the predicate `record_better_bound(E)` which takes the new value of E and posts this to the database. We assume that any answer to G fixes the value of E.

```
minimize(G, E) :-
     get_min_value(G, E, M),
     E = M,
     call(G).
```

```
get_min_value(G, E, _) :-
     apply_new_bound(E),
     once(G),
     record_better_bound(E),
     fail.
get_min_value(_, _, M) :- retract(bestbound(M)).

apply_new_bound(_).
apply_new_bound(E) :-
     retract(currentbound(B)),
     assert(bestbound(B)),
     E < B,
     apply_new_bound(E).

record_better_bound(E) :-
     (retract(bestbound(_)) -> true ; true),
     assert(currentbound(E)).
```

As an example of this program's execution consider the program

```
p(a, 10).
p(b, 5).
p(c, 8).
```

The most interesting part of the derivation tree for goal `minimize(p(X,Y), Y)` is the evaluation of the call `get_min_value(p(X,Y),Y, M)` which calculates the minimal value, M, for Y. The derivation tree for this call is shown in Figure 9.9. For brevity `get_min_value` is shortened to *gmv*, `apply_new_bound` to *anb*, `record_better_bound` to *rbb*, `currentbound` to *cbnd* and `bestbound` to *bbnd*. Derivation steps for database predicates and primitive constraints are omitted, but the effect of the database predicates is indicated next to the derivation step.

The key points in the exploration of the derivation tree are as follows. At the state, s_1,

$$\langle anb(Y), once(p(X,Y)), rbb(Y), fail \mid true \rangle$$

using the first rule for *anb* leads to the left branch and the state

$$\langle once(p(X,Y), rbb(Y), fail \mid true \rangle.$$

The first answer found for the goal `p(X,Y)` is $X = a \wedge Y = 10$. Next `rbb(Y)` tries to retract a `bestbound` fact (there is none) and asserts the `currentbound(10)` fact. The derivation now fails and backtracks to try the second rule for *anb* in the state s_1. This retracts `currentbound(10)` and asserts `bestbound(10)`, as well as adding the constraint $Y < 10$. For the child state, s_2,

$$\langle anb(Y), once(p(X,Y)), rbb(Y), fail \mid Y < 10 \rangle$$

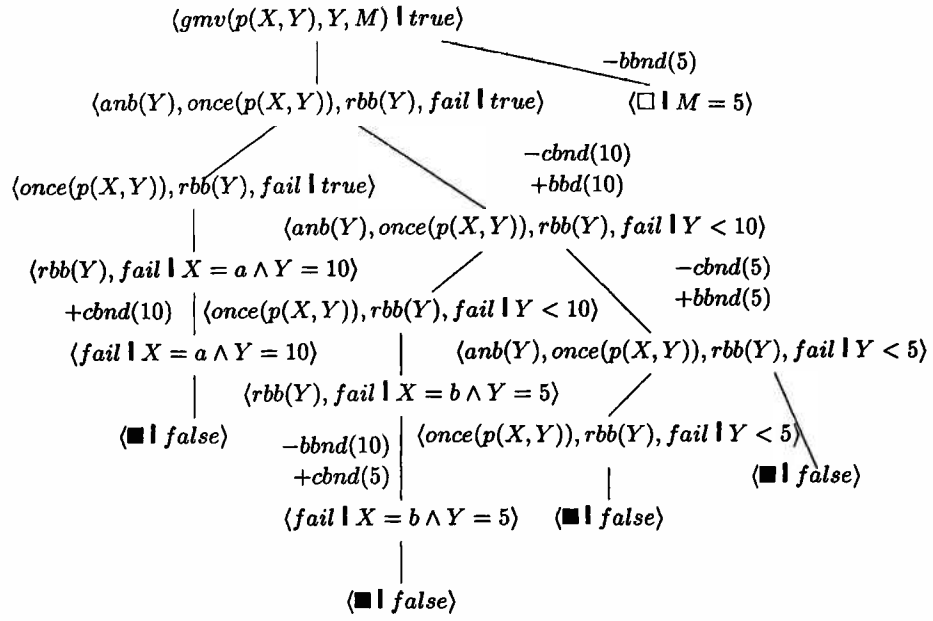

Figure 9.9 Using database predicates for minimize.

the first rule for *anb* leads to the state $\langle once(p(X,Y), rbb(Y), fail \mid Y < 10\rangle$. The derivation from here acts as before except that the first answer for `p(X,Y)` is now $X = b \wedge Y = 5$ because of the constraint $Y < 10$, so `bestbound(10)` is retracted and `currentbound(5)` is asserted and the derivation fails. The second choice for state s_2 retracts the `currentbound` fact and asserts it as a `bestbound`, and adds constraint $Y < 5$. At the state, s_3,

$$\langle anb(Y), once(p(X,Y)), rbb(Y), fail \mid Y < 5\rangle$$

the first rule for *anb* leads to $\langle once(p(X,Y), rbb(Y), fail \mid Y < 5\rangle$, which immediately fails, since there are no answers for $\langle p(X,Y) \mid Y < 5\rangle$. The second rule fails, because there is no `currentbound` fact in the database. Finally, the second rule for *gmv* is tried, which retracts the `bestbound(5)` fact, and constrains the minimal solution value M to 5.

9.8.4 Debugging Tools

No matter how high level the programming language, there comes a time when a seemingly correct program does not give the right answer and so must be debugged. Most CLP systems provide rather primitive debugging facilities based on the "Byrd Box" model of standard Prolog execution. In this model, program execution is understood in terms of predicate definitions. The programmer can specify which predicates they are interested in, and the debugger will provide information about

the evaluation of these predicates. The programmer can either specify that all predicates are of interest by commanding that the program should be *traced* or they can specify that a particular predicate is of interest by placing a *spypoint* on the predicate.

The debugger prints out information about a predicate of interest whenever one of the following four events occurs. The first event, *call*, occurs when the predicate is initially called. The debugger prints out the name of the predicate, the level that the call occurs at and some information about the current constraint store simplified in terms of the variables in the call. If the predicate is user-defined, the debugger will now detail when each rule in the definition is tried. If the call to the predicate succeeds, an *exit* event occurs. If the call to the predicate does not succeed, a *fail* event occurs. The remaining event occurs when, after an initial success, subsequent failure causes the predicate to be backtracked to. This causes a *redo* event which is treated very much like the initial call. Again the debugger will detail when different rules in the predicate are being tried. If another answer is found, another exit event occurs. If no other answer is found, a fail event occurs.

We illustrate the library functions of SICStus Prolog. Similar functions are defined and act in more or less the same way in other CLP systems. We advise readers to refer to the user's manual of their CLP system. The most important library functions for debugging are:

`trace`: Turns debugging on and indicates that all predicates are of interest and so their execution should be traced.

`notrace`: Turns the debugger and tracing off.

`debug`: Turns debugging on but does not place spypoints on any predicates.

`nodebug`: Turns the debugger off.

`spy(P)`: Places a spypoint on predicate P. Debugging should already be on.

`nospy(P)`: Removes a spypoint previously placed on predicate P.

`nospyall`: Removes all spypoints.

We demonstrate the debugging predicates on the simple goal

```
intersect([1,2], [0,2,4]).
```

where the user-defined constraint `intersect` succeeds if its two arguments are lists which have a common element. The code for `intersect` is given on page 333. The simplified derivation tree for the above goal is illustrated in Figure 9.7.

When using the debugging facilities, each time an event occurs in the execution a line is printed and the user is given the opportunity to modify the execution. The main options are: Enter/Return or "c", continues execution until the next event (regardless of whether the predicate has a spypoint); "l" which leaps to the next event for a predicate with a spypoint; "s" which skips all events until an *exit* or *fail* event for the current literal; "n" which continues the rest of the execution without debugging; "f" which causes an immediate failure for this literal; "r" which causes

the literal to be re-executed; and "a" which aborts execution. Tracing evaluation of the goal `intersect([1,2], [0,2,4])` produces the following session in which the user has typed Enter/Return at each prompt.

```
Call: intersect([1,2], [0,2,4]) ?
Call: member(X, [1, 2]) ?
Exit: member(1, [1, 2]) ?
Call: member(1, [0, 2, 4]) ?
Call: member(1, [2, 4]) ?
Call: member(1, [4]) ?
Call: member(1, []) ?
Fail: member(1, []) ?
Fail: member(1, [4]) ?
Fail: member(1, [2, 4]) ?
Fail: member(1, [0, 2, 4]) ?
Redo: member(1, [1, 2]) ?
Call: member(X, [2]) ?
Exit: member(2, [2]) ?
Exit: member(2, [1, 2]) ?
Call: member(2, [0, 2, 4]) ?
Call: member(2, [2, 4]) ?
Exit: member(2, [2, 4]) ?
Exit: member(2, [0, 2, 4]) ?
Exit: intersect([1,2], [0,2,4]) ?
```

Each line in the trace is related to a state in the derivation tree whose first literal corresponds to the printed literal. Different lines detail how this state is visited during the depth-first left-to-right traversal of the derivation tree. A line of the form `Call:` L details the first time the state with first literal L is visited. The line `Exit: member(1, [1,2])` indicates that the (first) answer $X = 1$ to the literal `member(X, [1, 2])` has been found. Now the second `member` literal is called. Each `Fail:` L line corresponds to moving back up the derivation tree after discovering that the derivation tree under the state in which literal L was rewritten is finitely failed. The `Redo: member(1, [1,2])` line corresponds to looking for a new answer to the literal `member(X, [1, 2])`, after already finding one, so this begins the traversal down the right branch from the second state. Ideally, the redo goal should be printed as `Redo: member(X, [1,2])` since the previous answer in which $X = 1$ is irrelevant for the purposes of finding the next answer. This behaviour occurs because it is easier to implement than is the idealized behaviour. The `Exit: member(2, [2])` line indicates that evaluation has found an answer, $X = 2$, to the call `member(X, [2])`. This is then returned as the second answer to the call `member(X, [1, 2])`. This leads to the call to the second member literal `Call: member(2, [0, 2, 4])`. Finding an answer to this literal leads to an answer to the overall goal.

As another example suppose the user types the goal

```
spy(intersect), debug, intersect([1,2], [0,2,4]).
```

One possible interaction is as follows:

```
Call: intersect([1,2], [0,2,4]) ? c
Call: member(X, [1, 2]) ? c
Exit: member(1, [1, 2]) ? c
Call: member(1, [0, 2, 4]) ? s
Fail: member(1, [0, 2, 4]) ? c
Redo: member(1, [1, 2]) ? s
Exit: member(2, [1, 2]) ? l
Exit: intersect([1,2], [0,2,4]) c ?
```

Because the `intersect` predicate is spied upon the user is notified of calls to this predicate. By creeping ("c") the next three events are reported even though `member` has no spypoints. The *fail* event is generated after skipping ("s") events from the event `Call: member(1, [0, 2, 4])`. Creeping gives the next event and skipping gives its (next) answer. Finally leaping ("l") skips events until a predicate with a spypoint is reached.

Debugging may be implemented using a source-to-source translation. For example, a version of `member` which outputs trace information if the fact `spy(member)` is in the database is:

```
member(X,L) :-
     (clause(spy(member),true) ->
          write("Call: "), write(member(X,L)), nl
     ; true),
     member1(X,L),
     (clause(spy(member),true) ->
          write("Exit: "), write(member(X,L)), nl
     ; true),
     (clause(spy(member),true) ->
          retry_member(X,L)
     ; true).
member(X,L) :-
     (clause(spy(member),true) ->
          write("Fail: "), write(member(X,L)), nl
     ; true),
     fail.

retry_member(_X,_L).
retry_member(X,L) :- write("Redo: "), write(member(X,L)), nl, fail.

member1(X, [X|_]).
member1(X, [_|R]) :- member(X,R).
```

If the fact `spy(member)` is in the database, the "Call" line is printed when `member` is first called, and the "Exit" line is printed when an answer is found. The auxiliary predicate `retry_member` succeeds the first time, doing nothing but leaving a choicepoint. This choicepoint is reached just before execution returns to a choicepoint for the `member1` call. The second rule for `retry_member` prints the "Redo" line and then fails so execution then returns to the choicepoint for the `member1` call. Because `retry_member` is called after `member1`, it misleadingly prints the literal with the previous answer. Finally, when there are no more answers to `member1`, the second rule for `member` prints the "Fail" line before failing.

9.9 Summary

In this chapter we have given examples of advanced programming techniques. The primary aim of these programming techniques is to allow the programmer to modify the underlying CLP system, either by extending the constraint solver or by specializing the standard CLP built-ins such as those for optimization. The ability to do this distinguishes CLP from constraint solving packages. CLP systems provide a programming language and environment in which it is possible, at least in theory, to implement any algorithm.

Extending the constraint solver to handle constraints for a different constraint domain, or new kinds of primitive constraints is a reasonably frequent activity in constraint programming. Unfortunately, if these new constraints are only defined using "vanilla" CLP, as described in Chapter 4, their allowed mode of usage is often quite restrictive. Fortunately, techniques such as dynamic scheduling and meta-programming allow the definition of robust and safe constraints that work correctly, though perhaps incompletely, for all modes of usage.

Dynamic scheduling allows the programmer to control selection of literals at run-time. This makes it easier to assure termination of goals.

Meta-programming is reasonably straightforward in constraint logic programming because terms can be used to represent programs, and the underlying execution method is simple. The advantage of meta-programming is complete control over the search strategy. Meta-programming thus provides an easy way to experiment with different search strategies which the user may implement later using more efficient built-ins such as those for dynamic scheduling.

Just as it is possible to extend the constraint solver, new optimization routines can be programmed. We gave two examples, one in which the program implemented a well-known optimization technique, and an example in which a different kind of optimization search strategy was defined.

Clearly negation is useful in modelling. Unfortunately, the kind of negation provided by CLP systems is limited in usefulness, because it only truly models "logical" negation when the negated goal has all of its variables fixed. However, one use of negation is to test the satisfiability of constraints without adding the constraint to the store.

9.10 Practical Exercises

The floor $\lfloor t \rfloor$ and ceiling $\lceil t \rceil$ functions are provided in SICStus Prolog as `floor(X)` and `ceiling(X)`. They are not provided directly in $CLP(\mathcal{R})$. The $CLP(\mathcal{R})$ built-in `floor(`t, f`)` holds if f is the integer floor of the fixed number t. One can define ceiling in terms of `floor`. The predicate `integer` can also be defined in terms of `floor`.

The real arithmetic constraint solvers for SICStus Prolog provide a built in branch and bound minimization solver. The built-in `bb_inf(`*Ints*, *Expr*, *Inf*`)` computes the infimum *Inf* of expression *Expr* given that all the variables in the list *Ints* must take integer values. So, for example, the problem of Example 9.1 could be solved using the goal:

```
problem([X,Y], L), bb_inf([X,Y], L, Best).
```

Both ECLiPSe and SICStus Prolog include facilities for dynamic scheduling. ECLiPSe provides a goal based delay construct `delay(`*Term*, *Goal*`)` which is similar to

`delay_until(nonvar(`X_1`) or ... or nonvar(`X_n`),` *Goal*`)`

where X_1 to X_n are the variables in the solved form of *Term* when the literal is selected. It also provides a predicate based delay construct. For instance, the declaration

```
delay append(X,Y,Z) if var(X), var(Z).
```

will delay calls to append if both the first and third arguments are variables. Declarations delay the execution of any literal which matches the declaration and where the body of the declaration (after `if`) succeeds. The user can write multiple delay declarations for one literal. For more details see the ECLiPSe User Manual.

SICStus Prolog provides the goal based delay construct `when(`*Cond*, *Goal*`)` which acts like `delay_until`. The condition can be `nonvar(X)`, which delays until X cannot take all possible values, `ground(X)` which delays until X is fixed, and `?=(X,Y)` which delays until X and Y are either definitely identical or cannot be equal. Conjunctions of delay conditions are written "*Cond*$_1$, *Cond*$_2$" and disjunctions as "*Cond*$_1$; *Cond*$_2$." For example, the not equals predicate can be defined by

```
ne(X,Y) :- when((ground(X),ground(Y)), not(X = Y)).
```

A better implementation of disequations is possible using the `?=(X,Y)` delay condition:

```
ne(X,Y) :- when(?=(X,Y), not(X = Y)).
```

SICStus Prolog also provides a (much more efficiently implemented) predicate based delay construct. The declaration

```
:- block append(-,?,-).
```

uses the pattern (-,?,-) to block the selection of append literals unless either the first or third argument is non-variable. Multiple patterns for the same predicate can be included in one declaration, and the declaration blocks if any evaluate to true. See the SICStus Prolog Users Manual for more information.

CLP($\mathcal{R}$), SICStus Prolog and ECLiPSe support the debugging facilities described in Subsection 9.8.4, as well as others. For details refer to the appropriate User Manual.

P9.1. Write code for the predicates `vs_mult`, `element` and `dotprod`.

P9.2. Use `solve_nr` to determine three values x (each further apart than 1) such that $x^3 - 2x^2 - 8x + 1$ is between -0.0001 and 0.0001.

P9.3. Write a program defining predicate `bisect(E, L, U, X)` to find a zero X of a nonlinear function f on a single variable (defined by predicate `f(X, F)` using the bisection method. That is given a range $[L..U]$ for which $f(L) \times f(U) < 0$, determine the value of $f(\frac{L+U}{2})$ and replace either the lower or upper bound with $\frac{L+U}{2}$ so that the condition continues to hold. The program should terminate when $U - L < E$. Use this program to check your answers to the above problem.

P9.4. (*) Build a better constraint solver for sequence constraints by incorporating reasoning about lengths. We can represent a sequence as a pair $s(list, length)$ where *list* is a list representing the sequence and *length* is its length. The sequence constraint $S3 = S1 :: S2$ can be translated as:

```
N1 ≥ 0, N2 ≥ 0, N3 = N1 + N2, seq_append(L1, L2, L3, N1).
```

where Si is the pair $s(Li, Ni)$, $1 \leq i \leq 3$ and the `seq_append` predicate is defined below

```
seq_append([], L2, L2, 0).
seq_append([A|L1], L2, [A|L3], N) :-
    N = N0+1, N0 ≥ 0, seq_append(L1, L2, L3, N0).
```

Determine translations for $S \neq \epsilon$ and $S = nil$.
Try the goal corresponding to the sequence constraint $S2 \neq \epsilon \wedge U = nil \wedge T = S1 :: S2 \wedge U = T :: S3$.

P9.5. Reexamine the problem of Exercise P5.2. Given the river flows at 2 m/s during the flooding of the wet season, and the traveller rows at 1.5 m/s across a 12 m wide river and has to end up exactly 20 m downstream from the setting off position, find the angle θ at which she must row.
The built-in nonlinear constraint solver is not powerful enough to solve these constraints. Convert the problem into an expression e in terms of θ which is zero if the constraints hold. Use the Newton-Raphson method and symbolic differentiation to determine for what value of θ the constraints hold.
What happens if the river is 24 m wide?

P9.6. Write a program for `not_append(X,Y,Z)` which holds if Z is not the list obtained by appending X and Y. Assume each of X, Y and Z is a list.

P9.7. Write a program for `simplify(E0,E)` which simplifies an arithmetic expression $E0$ to another E (both represented as trees). For example the expression $(X - X) \times Y + Z - (2Z + -T)$ might be simplified to $-Z + T$.

P9.8. Extend the delay based solver for the Boolean constraints `bnot(X,Y)` and `and(X,Y,Z)` to handle `or(X,Y,Z)`, `xor(X,Y,Z)`, `if(X,Y,Z)`, and `iff(X,Y,Z)`. Ensure that the predicates are deterministic whenever their delay conditions are enabled.

P9.9. Using dynamic scheduling, write a definition for the user-defined constraint `length(L,N)` which holds if N is the length of list L. Ensure that, with this definition, none of the following goals has an infinite derivation:

(a) `length(L,N), L = [1,2], fail`

(b) `length(L,N), N = 3, fail`

(c) `length(L,N), N ≤ 3, append([1,2,3,4|_], A, L)`

(d) `length(L1,N1), length(L2,N2), N1+N2≤2, append(L1,L2,[1,2,3])`.

P9.10. (*) Using dynamic scheduling and negation, extend the constraint solver to handle the constraint `sqr(X,Y)` which holds when $Y = X^2$. Attempt to make the handling deterministic but as complete as possible, so that the following goals all succeed:

(a) `sqr(X,Y), X = 1, Y = 1`

(b) `Y = -3, not(sqr(X,Y))`

(c) `sqr(X,Y), var(X), X = 1, arithmetic(Y)`

(d) `sqr(X,Y), Y = 4, var(X)`

(e) `sqr(X,Y), X ≥ 1, Y = 4, arithmetic(X)`

(f) `sqr(X,Y), X ≤ 1, Y = 4, nonvar(X)`.

P9.11. (*) Use the meta-interpreter of Section 9.7 which handles `append` specially to execute the goal `derive([append(X,Y,Z), append(Y,X,Z)])`. Explain the result.

P9.12. (*) The `is` built-in of Prolog provides arithmetic calculation. The literal t_1 `is` t_2 will execute only if t_2 is a ground arithmetic expression, in which case it will evaluate the expression (obtaining n) and attempt to equate t_1 to n. For example, the goal `Y = 3, X is Y * Y + 4` succeeds with $Y = 3 \wedge X = 13$. We can use this to implement a limited form of arithmetic solving.
Modify the meta-interpreter of Section 9.7 to handle the constraint $Z = X + Y$, represented as `plus(X, Y, Z)` using local propagation. For example, if X and Z are fixed, then Y can be calculated by $Y = Z - X$ using the rule:

```
plus(X, Y, Z) :- ground(X), ground(Z), Y is Z - X.
```

Execute the goal

```
derive([plus(X,Y,Z), plus(Z,Y,U), plus(V,3,U), Z = 3, Z = 1]).
```

Extend the meta-interpreter to handle `mult(X,Y,Z)`.

P9.13. Write a higher-order predicate `solutions(G, X, L)` which collects as a list L all the values of X occurring in answers of the goal G. You will need to use `call` as well as the database built-ins to do so.

P9.14. (*) Consider the blocks world example of Section 1.5.3. We can represent the different blocks in the example world $[yc, gc, rs, gp, rp]$ using integers $[1, 2, 3, 4, 5]$. We can represent a value for a variable in the blocks world obj_1 *on* obj_2 using the number $obj_1 + 5 \times obj_2$ where if obj_2 is the $floor$ then its value is 0. Write a $CLP(FD)$ program to define predicates for the constraints `on(X,Y)`, `floor(X)`, `equal(X, Y)`, `not_equal(X, Y)`, `red(X)`, `yellow(X)`, `not_green(X)` and `pyramid(X)`. Use it to find answers to the sample constraints given in Section 1.5.3 and Exercise 1.8.

9.11 Notes

Newton-Raphson is a well known method for finding the roots of differentiable functions. The branch and bound method for integer linear programming is due to Dakin. For more detailed descriptions of branch and bound see, for example, Papadimitriou and Steiglitz [101] or Schrijver [116]. Optimistic partitioning is amenable to straightforward parallelization. Some experiments making use of it are described in [103].

Negation in CLP is inherited from Prolog. This form of negation, negation by finite failure [28], is quite unsuited to constraint logic programming, since, in a typical CLP program, it is unlikely that negative goals only have fixed arguments. Furthermore, this form of negation interacts badly with incomplete constraint solvers. More suitable forms of negation are known for constraint logic programming, so called "constructive negation" [25, 128], but efficient implementation of these ideas is not straightforward so current systems do not support them.

The limitations of a fixed literal selection strategy were recognised early in the development of logic programming languages. Absys1 [47] a precursor to Prolog provided dynamic scheduling as did the logic programming languages IC-Prolog [30], Prolog-II [33] and MU-Prolog [96]. For example predicate based delay is provided by `wait` declarations for MU-Prolog, `when` declarations for NU-Prolog, and `block` declarations for SICStus Prolog. Goal based delay declarations are provided by `freeze` subgoals in Prolog-II, and `freeze`, and `when` subgoals in SICStus Prolog. The use of dynamic scheduling for writing constraint solvers appears in the paper of Kawamura et al [78]. Now most logic programming languages and constraint logic programming languages provide dynamic scheduling. However, the dynamic scheduling delay conditions provided in CLP systems are inherited from Prolog and,

therefore, may not provide all of the functionality a constraint programmer desires.

Meta-programming in CLP is inherited from Prolog. Hence some of the meta-programming facilities for CLP are clumsy and unnatural. For a discussion of meta-programming in Prolog see [125, 1], while various approaches to meta-programming in CLP are discussed in [63, 88, 87].

Library functions for CLP languages are inherited from Prolog. Many of them are somewhat ill-defined for constraint domains other than tree constraints. The definition of the `variables` predicate is based on a program given in the SICStus Prolog User's Manual [58].

Debugging constraint programs can be a difficult task since the affect of the constraint solver may be hard to determine. Debugging facilities for CLP are inherited from Prolog and are less than adequate for constraint domains other than the tree constraint domain. Specific extensions for other constraint domains are not standardized at present. The reader is referred to the survey by Ducassé and Noyé of debugging for Prolog-like languages [45]. A source-to-source transformation for tracing is described in this.