4 Constraint Logic Programs

In the first part of this book we examined various constraint domains, including real arithmetic constraints, tree constraints and finite domains, as well as three fundamental operations involving constraints—satisfaction, simplification and optimization. We have also seen how constraints can be used to model the behaviour of physical objects and processes.

In this part we introduce the constraint logic programming (CLP) paradigm. In this chapter we define constraint logic programs and their evaluation mechanism and in subsequent chapters we will investigate programming techniques. Finally we will examine the implementation of CLP systems.

Constraint logic programming languages are parametric in the choice of the underlying constraint domain and the solver and simplifier for that domain. In essence, constraint logic programming languages provide only one programming construct—"rules." Rules allow programmers to define their own constraints in terms of the underlying constraint domain of the CLP language. Somewhat surprisingly, this simple extension leads to a fully-fledged programming language based on constraints in which a program is simply a collection of rules.

Rules are used to evaluate "goals" given by the user. This is done by repeatedly using the rules to replace the definition of user-defined constraints in the goal until only primitive constraints are left. The resulting constraint is the "answer" to the goal. This process of rewriting is called a "derivation."

4.1 User-Defined Constraints

In most of the examples so far of modelling an object or process with a constraint, the constraint could be partitioned into two parts: a general description of the object or process being modelled, and specific information detailing the situation at hand. Since many questions use the same general constraint description, it is useful to have a mechanism which allows a constraint description to be reused in different problems. In general, we would like users, that is programmers, to be able to define their own problem specific constraints in terms of the underlying primitive constraints. *Rules* enable this.

Example 4.1

Consider the circuit described in Figure 2.1. If we are only interested in its behaviour with respect to the overall voltage V, overall current I and the two resistance values $R1$ and $R2$, then we can introduce the new constraint,

```
parallel_resistors(V, I, R1, R2),
```

which describes this behaviour. The new constraint is defined using a *rule* as follows.

```
parallel_resistors(V,I,R1,R2) :- V = I1*R1, V = I2*R2, I1+I2 = I.
```

The rule says that θ is a solution to `parallel_resistors(V, I, R1, R2)` if θ is a partial solution to the constraint

$$V = I1 \times R1 \wedge V = I2 \times R2 \wedge I1 + I2 = I.$$

Note that we shall use `typewriter` font for defining rules and programs and for referring to user-defined constraints in text. Constraints (from the constraint domain) are written in italics as usual. This means that we shall sometimes refer to the same variable, for example V using two different fonts V and `V`.

Now, rather than writing the conjunction of arithmetic constraints describing parallel resistors each time we need them, we can use this new constraint. For instance, suppose we are interested in the behaviour of two parallel resistors of values 10Ω and 5Ω. The constraint,

$$\texttt{parallel\_resistors}(V, I, R1, R2) \wedge R1 = 10 \wedge R2 = 5,$$

models this behaviour. Similarly, if we are studying such a circuit in which the voltage is 10V and the resistors are only known to have the same value, it can be described by

$$\texttt{parallel\_resistors}(10, I, R, R).$$

Intuitively, the meaning of the above statement can be understood by using the defining rule to replace an instance of `parallel_resistors` by the constraint on the right hand side of the rule. Thus the statement above represents the constraint

$$10 = I1 \times R \wedge 10 = I2 \times R \wedge I1 + I2 = I.$$

Rules enable the programmer to define new constraints which model the behaviour of a particular problem. We call these new constraints user-defined constraints.

Definition 4.1

A *user-defined constraint* is of the form $p(t_1, \ldots, t_n)$ where p is a n-ary *predicate* and $t_1, \ldots, t_n$ are expressions from the constraint domain. We shall use strings beginning with a lower case letter and made up of alphabetic characters and digits and the underscore "_" character to name predicates.

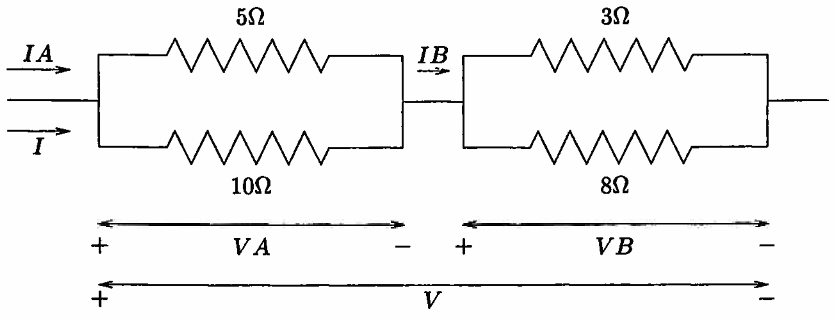

Figure 4.1 Two parallel resistors in series.

A *literal* is either a primitive constraint or a user-defined constraint.

A *goal*, G, is a sequence of literals. That is, G has the form $L_1, L_2, \ldots, L_m$, where $m \geq 0$ and each L_i is a literal. In the case $m = 0$, the goal is said to be *empty* and is represented by "$\square$."

A *rule*, R, is of the form A :- B where A is a user-defined constraint and B is a goal. A is called the *head* of R and B is called the *body* of R.

A *fact*, is a rule with the empty goal as body, A :- $\square$, and is simply written as A.

A *(constraint logic) program* is a sequence of rules.

The *definition* of a predicate p in a program P is the sequence of rules appearing in P which have a head involving predicate p.

In Example 4.1 above, `parallel_resistors` is a predicate, which is defined by a single rule:

```
parallel_resistors(V,I,R1,R2) :- V = I1*R1, V = I2*R2, I1+I2 = I.
```

The head of this rule is `parallel_resistors(V, I, R1, R2)` and the body is `V = I1*R1, V = I2*R2, I1+I2 = I`. Another example of a user-defined constraint is the literal `parallel_resistors(10, I, R, R)`.

Intuitively, rules in a constraint logic program can be viewed as "macro" definitions. However, we need to be careful when defining the mechanism of rewriting, in order to avoid problems with the scope of local names in the body of the rule.

Example 4.2

Consider the goal representing two parallel resistors in sequence as shown in Figure 4.1:

```
    parallel_resistors(VA, IA, 10, 5),
    parallel_resistors(VB, IB, 8, 3),
    VA + VB = V, I = IB, I = IA.
```

Simply replacing each predicate by the body of its definition gives

```
    VA = I1 * 10, VA = I2 * 5, IA = I1 + I2,
    VB = I1 * 8, VB = I2 * 3, IB = I1 + I2,
    VA + VB = V, I = IB, I = IA.
```

As this goal only involves primitive constraints, we can treat it as a constraint by replacing the commas by conjunction ($\wedge$). The resulting constraint has only one solution, namely,

$$\{V \mapsto 0, VA \mapsto 0, VB \mapsto 0, IA \mapsto 0, IB \mapsto 0, I \mapsto 0, I1 \mapsto 0, I2 \mapsto 0\}.$$

This is incorrect since it does not model the actual behaviour of the circuit in Figure 4.1. The problem arises because "macro" replacement has confused the two uses of the local variables $I1$ and $I2$.

The solution to this problem is to use new variables each time we use a rule to replace a user-defined constraint within a goal. To do this we make use of "renamings". Renamings were introduced in Section 1.6, but now we extend their definition from constraints to other syntactic objects.

Definition 4.2
A ***syntactic object*** is a constraint, user-defined constraint, rule or goal.
The result of ***applying*** a renaming ρ to a syntactic object o, written $\rho(o)$, is the expression obtained by replacing each variable x in o by $\rho(x)$.
A syntactic object o is a ***variant*** of syntactic object o' if there is a renaming ρ such that $\rho(o) \equiv o'$.

It is convenient to be able to use rules in which the arguments in the head of the rule are not distinct variables. For example, the parallel resistor definition can be simplified by writing it as

```
parallel_resistors(V, I1+I2, R1, R2) :- V = I1 * R1, V = I2 * R2.
```

To handle such definitions correctly we define the rewriting process as follows.

Definition 4.3
Let goal G be of the form

$$L_1, \ldots, L_{i-1}, L_i, L_{i+1}, \ldots, L_m$$

where L_i is the user-defined constraint $p(t_1, \ldots, t_n)$ and let rule R be of the form

$$p(s_1, \ldots, s_n) \text{ :- } B.$$

A ***rewriting*** of G at L_i by R using ρ is the goal

$$L_1, \ldots, L_{i-1}, t_1 = \rho(s_1), \ldots, t_n = \rho(s_n), \rho(B), L_{i+1}, \ldots, L_m$$

where ρ is a renaming chosen so that the variables in $\rho(R)$ do not appear in G.

Example 4.3
Consider the goal

```
    parallel_resistors(VA, IA, 10, 5),
    parallel_resistors(VB, IB, 8, 3),
    VA + VB = V, I = IB, I = IA.
```

from before. Rewriting the first user-defined constraint with the rule

```
parallel_resistors(V,I,R1,R2)  :-  V = I1*R1, V = I2*R2, I1+I2 = I.
```

and the renaming $\{V \mapsto V', I \mapsto I', R1 \mapsto R1', R2 \mapsto R2', I1 \mapsto I1', I2 \mapsto I2'\}$ gives

```
VA = V', IA = I', 10 = R1', 5 = R2',
V' = I1' * R1', V' = I2' * R2', I1' + I2' = I',
parallel_resistors(VB, IB, 8, 3),
VA + VB = V, I = IB, I = IA.
```

Rewriting the remaining user-defined constraint using the same rule and the renaming $\{V \mapsto V'', I \mapsto I'', R1 \mapsto R1'', R2 \mapsto R2'', I1 \mapsto I1'', I2 \mapsto I2''\}$ gives

```
VA = V', IA = I', 10 = R1', 5 = R2',
V' = I1' * R1', V' = I2' * R2', I1' + I2' = I',
VB = V'', IB = I'', 8 = R1'', 3 = R2'',
V'' = I1'' * R1'', V'' = I2'' * R2'', I1'' + I2'' = I'',
VA + VB = V, I = IB, I = IA.
```

Since this goal only involves primitive constraints we can consider it a constraint by replacing commas by $\wedge$. The overall behaviour of the circuit is the relationship between variables V and I. Simplifying the resulting constraint in terms of the variables V and I, gives the answer $V = (26/3) \times I$.

Using user-defined constraints we can write goals that naturally and concisely specify large, complicated constraints. The specified constraint can be found by repeatedly applying rewriting steps until the goal consists entirely of primitive constraints. This resulting constraint is the answer to the goal. However, the use of renamings in rewriting introduces many new variables which are almost certainly not of interest to the constraint programmer writing the goal. For this reason, after rewriting is finished, the resulting constraint should be simplified with respect to the variables in the original goal, as we have done in the previous examples.

4.2 Programming with Rules

User-defined constraints are more powerful than the example of the parallel resistor suggests. The definition of rules allows predicates to be defined in terms of other predicates. In particular, this allows a predicate to be defined *recursively* in terms of itself. It also allows a predicate to be defined by more than one rule. This enables us to model choice and case statements.

The following example illustrates how multiple rules can be used to capture different cases in the definition of a predicate. They can also be used to give a natural modelling of choice.

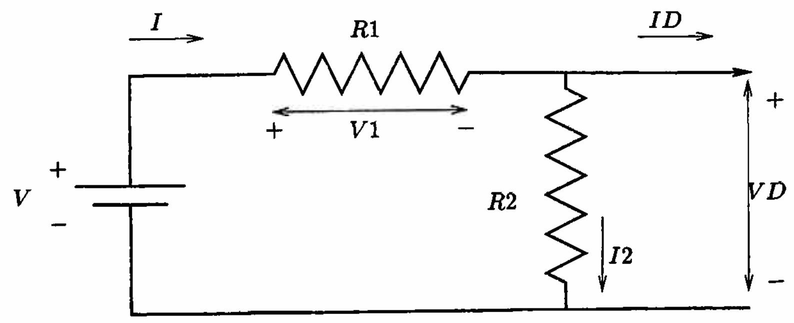

Figure 4.2 A voltage divider circuit.

Example 4.4
A voltage divider is used to create an output voltage satisfying some constraints from the available resistor and voltage cell components. Consider the problem of building a voltage divider circuit of the form shown in Figure 4.2. The constraint describing the circuit behaviour can be captured by the rule

```
voltage_divider(V, I, R1, R2, VD, ID) :-
    V1 = I * R1, VD = I2 * R2, V = V1 + VD, I = I2 + ID.
```

Suppose we wish to build a circuit which has divider voltage VD in the range 5.4V to 5.5V when the divider current ID is 0.1A. These conditions are simply expressed by the constraint $5.4 \leq VD \land VD \leq 5.5 \land ID = 0.1$. In addition, the circuit components must be chosen from those available in our laboratory. The cell must be 9V or 12V, and the resistors must be $5\Omega, 9\Omega$ or 14Ω.

We can use multiple rules to express the choice of the cell and the choice of the resistors. Each rule is considered to be one possible definition of the predicate, and the process of rewriting a goal chooses which rule to apply. The constraint that the cell must be 9V or 12V is expressed by the facts:

```
cell(9).
cell(12).
```

Similarly the constraint that resistors should be 5Ω, 9Ω or 14Ω is expressed as:

```
resistor(5).
resistor(9).
resistor(14).
```

The complete voltage divider problem is specified by the goal

```
    voltage_divider(V, I, R1, R2, VD, ID),
    5.4 ≤ VD, VD ≤ 5.5, ID = 0.1,
    cell(V), resistor(R1), resistor(R2).
```

One possible sequence of rewritings for this goal uses the rule for predicate `voltage_divider` to rewrite `voltage_divider(V, I, R1, R2, VD, ID)`, the first rule for predicate `cell` to rewrite `cell(V)` and (two copies of) the first rule for `resistor` to rewrite `resistor(R1)` and `resistor(R2)`. The resulting constraint (after replacing comma's by conjunction) is

$$V = V' \wedge I = I' \wedge R1 = R1' \wedge R2 = R2' \wedge VD = VD' \wedge ID = ID' \wedge$$
$$V1' = I' * R1' \wedge VD' = I2' * R2' \wedge V' = V1' + VD' \wedge I' = I2' + ID' \wedge$$
$$5.4 \le VD \wedge VD \le 5.5 \wedge ID = 0.1 \wedge V = 9 \wedge R1 = 5 \wedge R2 = 5.$$

Unfortunately, this is unsatisfiable and so it does not give a solution to the problem of designing the voltage divider.

Another possible sequence of rewritings for this goal, uses the rule for predicate `voltage_divider` to rewrite the `voltage_divider(V, I, R1, R2, VD, ID)`, the first rule for predicate `cell` to rewrite `cell(V)`, the first rule for `resistor` to rewrite `resistor(R1)` and the second rule for `resistor(R2)`. The resulting constraint is

$$V = V' \wedge I = I' \wedge R1 = R1' \wedge R2 = R2' \wedge VD = VD' \wedge ID = ID' \wedge$$
$$V1' = I' * R1' \wedge VD' = I2' * R2' \wedge V' = V1' + VD' \wedge I' = I2' + ID' \wedge$$
$$5.4 \le VD \wedge VD \le 5.5 \wedge ID = 0.1 \wedge V = 9 \wedge R1 = 5 \wedge R2 = 9.$$

This is satisfiable. Simplifying the constraint on to the variables we are interested in, namely the values for the components in the design V, $R1$ and $R2$, gives the constraint $V = 5 \wedge R1 = 5 \wedge R2 = 9$. From this we can see how to design the voltage divider using the available components.

Overall there are 18 different possible rewritings of the original goal, of which all but one is unsatisfiable. This illustrates why we may need to search through possible rewritings of the goal in order to find a solution.

Rules also allow us to define predicates recursively, that is in terms of themselves. This ability vastly increases the expressive power of constraint programs, and it is what makes CLP languages true programming languages rather than just constraint definition languages.

Consider the factorial function which is defined by

$$N! = \begin{cases} 1 & \text{if } N = 0 \\ N \times (N-1)! & \text{if } N \ge 1. \end{cases}$$

We can write a predicate `fac(N,F)` which is true if F is $N!$ by

```
fac(0,1).                                (R1)
fac(N, N * F)  :-  N ≥ 1, fac(N-1,F).    (R2)
```

Note that we have labelled each rule with a name in parentheses. This is solely for reference in the text and is not part of the program. Each rule captures a different case in the definition of the function. The rule $R1$ expresses the base case, while the rule $R2$ naturally expresses the recursive case.

We can use this predicate to compute the factorial function as follows. Imagine we wish to compute the factorial of 2. We start with the goal `fac(2,X)` and repeatedly rewrite it as follows:

$$\begin{array}{c} fac(2,X) \\ \Downarrow R2 \\ 2=N,\ X=N\times F,\ N\geq 1,\ fac(N-1,F) \\ \Downarrow R2 \\ 2=N,\ X=N\times F,\ N\geq 1,\ N-1=N',\ F=N'\times F',\ N'\geq 1,\ fac(N'-1,F') \\ \Downarrow R1 \\ 2=N, X=N\times F, N\geq 1, N-1=N', F=N'\times F', N'\geq 1, N'-1=0, F'=1. \end{array}$$

In each step the rule used is written beside the arrow. We can simplify the final goal, treated as a constraint, with respect to the variable X and we obtain the expected answer, $X=2$.

Now consider what would have happened if we had selected rule $R2$ instead of $R1$ in the last rewrite step. In this case we obtain the goal:

$$\begin{array}{l} 2=N,\ X=N\times F,\ N\geq 1,\ N-1=N',\ F=N'\times F',\ N'\geq 1, \\ N'-1=N'', F'=N''\times F'', N''\geq 1, fac(N''-1,F''). \end{array}$$

Although the conjunction of the primitive constraints appearing in the goal is unsatisfiable, we can still rewrite the predicate `fac`. In fact, by always choosing $R2$ we can go on rewriting forever. However, such a rewriting does not provide any more information as the conjunction of primitive constraints appearing in each goal will remain unsatisfiable since rewriting will only add constraints to an already unsatisfiable conjunction.

4.3 Evaluation

Examples in the last section suggest that, when rewriting a goal, we check that the primitive constraints already collected are satisfiable. If they are not, rewriting can stop, since further rewriting can only lead to an unsatisfiable constraint. We now give an evaluation method for constraint logic programs, formalising this idea.

The evaluation method is based on a *derivation*. At each step in the derivation a literal is processed. If the literal is a user-defined constraint, it is rewritten as before. If it is a primitive constraint it is collected in the *constraint store*. The new constraint store is then tested for satisfiability. If it is unsatisfiable, rewriting stops immediately. Otherwise, the process continues until there are no literals left. At each step in the derivation we therefore have the constraint store, which is a conjunction of primitive constraints, and the remaining goal, which is a sequence of literals. We call this a *state* of computation.

Definition 4.4
A *state* is a pair written $\langle G \mid C \rangle$ where G is a goal and C is a constraint. C is called the *constraint store.*
A *derivation step* from $\langle G_1 \mid C_1 \rangle$ to $\langle G_2 \mid C_2 \rangle$, written $\langle G_1 \mid C_1 \rangle \Rightarrow \langle G_2 \mid C_2 \rangle$, is defined as follows. Let G_1 be the sequence of literals

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

There are two cases.

1. L_1 is a primitive constraint. Then C_2 is $C_1 \land L_1$ and, if $solv(C_2) \equiv false$, G_2 is the empty goal; otherwise G_2 is $L_2, \ldots, L_m$.
2. L_1 is a user-defined constraint. Then C_2 is C_1 and G_2 is a rewriting of G_1 at L_1 by some rule R in the program using a renaming ρ such that the variables in $\rho(R)$ are different from those in C_1 and G_1. If there is no rule defining the predicate of L_1 then C_2 is $false$ and G_2 is the empty goal.

A *derivation* for state $\langle G_0 \mid C_0 \rangle$ is a sequence of states

$$\langle G_0 \mid C_0 \rangle \Rightarrow \langle G_1 \mid C_1 \rangle \Rightarrow \langle G_2 \mid C_2 \rangle \Rightarrow \ldots$$

such that for each $i \geq 0$ there is a derivation step from $\langle G_i \mid C_i \rangle$ to $\langle G_{i+1} \mid C_{i+1} \rangle$ and the renamed rules used in each derivation step do not contain variables which occur earlier in the derivation.
If G is a goal then a *derivation for G* is a derivation for the state $\langle G \mid true \rangle$.

The different choices of renaming that are possible in a derivation step essentially make no difference since the names of the variables are irrelevant to the constraint solver, which we assume is well-behaved. The different choices of rule that are possible in rewriting a user-defined constraint may, however, lead to different derivations.

Example 4.5
Recall the earlier definition of the factorial predicate. One derivation from the goal `fac(2,X)` is illustrated in Figure 4.3.

In each step, if a user-defined constraint is rewritten, the rule used is shown beside the arrow. No further derivation steps are possible because the goal has become empty. This corresponds to a complete rewriting. The final constraint store is an answer to the initial state. The intermediate variables are not of interest. Thus the answer we would really like is the final constraint store simplified with respect to the variables in the initial goal, in this case X. This is simply $X = 2$, which is thus an "answer" to `fac(2,X)`.

A derivation can continue until the goal becomes empty. A derivation that can no longer continue can be either successful or failed.

$$\begin{array}{c}
\langle fac(2,X) \mid true\rangle \\
\Downarrow R2 \\
\langle 2 = N, X = N \times F, N \geq 1, fac(N-1,F) \mid true\rangle \\
\Downarrow \\
\langle X = N \times F, N \geq 1, fac(N-1,F) \mid 2 = N\rangle \\
\Downarrow \\
\langle N \geq 1, fac(N-1,F) \mid 2 = N \land X = N \times F\rangle \\
\Downarrow \\
\langle fac(N-1,F) \mid 2 = N \land X = N \times F \land N \geq 1\rangle \\
\Downarrow R2 \\
\langle N-1 = N', F = N' \times F', N' \geq 1, fac(N'-1,F') \mid 2 = N \land X = N \times F \land N \geq 1\rangle \\
\Downarrow \\
\langle F = N' \times F', N' \geq 1, fac(N'-1,F') \mid 2 = N \land X = N \times F \land N \geq 1 \land N-1 = N'\rangle \\
\Downarrow \\
\langle N' \geq 1, fac(N'-1,F') \mid 2 = N \land X = N \times F \land N \geq 1 \land N-1 = N' \land F = N' \times F'\rangle \\
\Downarrow \\
\langle fac(N'-1,F') \mid 2 = N \land X = N \times F \land N \geq 1 \land N-1 = N' \land F = N' \times F' \land N' \geq 1\rangle \\
\Downarrow R1 \\
\langle N'-1 = 0, F' = 1 \mid 2 = N \land X = N \times F \land N \geq 1 \land N-1 = N' \land \\
F = N' \times F' \land N' \geq 1\rangle \\
\Downarrow \\
\langle F' = 1 \mid 2 = N \land X = N \times F \land N \geq 1 \land N-1 = N' \land F = N' \times F' \land \\
N' \geq 1 \land N'-1 = 0\rangle \\
\Downarrow \\
\langle \Box \mid 2 = N \land X = N \times F \land N \geq 1 \land N-1 = N' \land F = N' \times F' \land \\
N' \geq 1 \land N'-1 = 0 \land F' = 1\rangle
\end{array}$$

Figure 4.3 A successful derivation for `fac(2,X)`.

Definition 4.5
A *success state* is a state $\langle G \mid C\rangle$ where G is the empty goal and $solv(C) \not\equiv false$.
A *fail state* is a state $\langle G \mid C\rangle$ where G is the empty goal and $solv(C) \equiv false$.
A derivation

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

is *successful* if $\langle G_n \mid C_n\rangle$ is a success state. The constraint $simpl(C_n, vars(\langle G_0 \mid C_0\rangle))$ is said to be an *answer* to the state $\langle G_0 \mid C_0\rangle$.
If G is a goal, then an *answer* for G is an answer to the state $\langle G \mid true\rangle$.
A derivation

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

is *failed* if $\langle G_n \mid C_n \rangle$ is a fail state.

The derivation in Figure 4.3 is an example of a successful derivation. From the definition of a derivation step, failed derivations can occur in two ways. Either a primitive constraint is added to the current constraint store making it unsatisfiable or a user-defined constraint is selected which has no rules defining it. Below we give examples of both kinds of failed derivation.

Consider the factorial predicate. One failed derivation for the goal `fac(2,X)` is:

$$\begin{array}{c} \langle fac(2,X) \mid true \rangle \\ \Downarrow R1 \\ \langle 2 = 0, X = 1 \mid true \rangle \\ \Downarrow \\ \langle \Box \mid 2 = 0 \rangle \end{array}$$

Now consider the goal `X = 2, nodefinition(X)` where `nodefinition` is a predicate without any defining rules. There is only one derivation for this goal and it is failed.

$$\begin{array}{c} \langle X = 2, nodefinition(X) \mid true \rangle \\ \Downarrow \\ \langle nodefinition(X) \mid X = 2 \rangle \\ \Downarrow \\ \langle \Box \mid false \rangle \end{array}$$

In practice the second type of failed derivation illustrated above rarely occurs. It is usually considered a programming error, much like calling a procedure without first defining it in a traditional programming language.

4.4 Derivation Trees and Finite Failure

Because there may be multiple derivations for a goal, evaluation of the goal may need to search to find a successful derivation. In order to do this, the CLP system explores a tree, called the *derivation tree*, which contains all the derivations for a particular goal. The derivation tree is a tree in which each path from the top of the tree is a derivation. Branches occur in the tree whenever there is a choice of rule to rewrite a user-defined constraint with.

Definition 4.6
A *derivation tree* for a goal G and program P is tree with states as nodes. It is constructed as follows. The root of the tree is the state $\langle G \mid true \rangle$. The children of each state $\langle G_i \mid C_i \rangle$ in the tree are those states which can be reached in a single derivation step from that state. A state which has two or more children is called a *choicepoint*.

It follows from this definition that, if the first literal in G_i is a primitive constraint, the node $\langle G_i \mid C_i \rangle$ has a single child. However, if the first literal is a user-defined constraint then there is a child for each rule in the definition of the literal. These children are ordered from left-to-right according to the relative ordering of the rules in the program.

A derivation tree represents all of the derivations from a goal. Different branches arise at choicepoints because there is a choice of rule with which to rewrite a user-defined constraint. A successful derivation is represented in a derivation tree by a path from the root to a success state. A failed derivation is represented in a derivation tree by a path from the root to a fail state. To help distinguish successful and failed derivations, we shall use "■" to represent the empty goal in a fail state and "□" to represent the empty goal in a success state.

Again recall the definition of the factorial predicate. The derivation tree for the goal `fac(2,X)` is shown in Figure 4.4. We have labelled arcs arising when a user-defined constraint is rewritten by the rule used in the rewriting. Note that states resulting from rewriting with rule $R1$ appear to the left of states resulting from rewriting $R2$, since $R1$ occurs before $R2$ in the program. The constraints in the derivation tree are:

$C_0 : true$

$C_1 : 2 = 0$

$C_2 : 2 = N$

$C_3 : 2 = N \land X = N \times F$

$C_4 : 2 = N \land X = N \times F \land N \geq 1$

$C_5 : 2 = N \land X = N \times F \land N \geq 1 \land N - 1 = 0$

$C_6 : 2 = N \land X = N \times F \land N \geq 1 \land N - 1 = N'$

$C_7 : 2 = N \land X = N \times F \land N \geq 1 \land N - 1 = N' \land F = N' \times F'$

$C_8 : 2 = N \land X = N \times F \land N \geq 1 \land N - 1 = N' \land F = N' \times F' \land N' \geq 1$

$C_9 : 2 = N \land X = N \times F \land N \geq 1 \land N - 1 = N' \land F = N' \times F' \land N' \geq 1 \land$
$N' - 1 = 0$

$C_{10} : 2 = N \land X = N \times F \land N \geq 1 \land N - 1 = N' \land F = N' \times F' \land N' \geq 1 \land$
$N' - 1 = 0 \land F' = 1$

$C_{11} : 2 = N \land X = N \times F \land N \geq 1 \land N - 1 = N' \land F = N' \times F' \land N' \geq 1 \land$
$N' - 1 = N''$

$C_{12} : 2 = N \land X = N \times F \land N \geq 1 \land N - 1 = N' \land F = N' \times F' \land N' \geq 1 \land$
$N' - 1 = N'' \land F' = N'' \times F''$

$C_{13} : 2 = N \land X = N \times F \land N \geq 1 \land N - 1 = N' \land F = N' \times F' \land N' \geq 1 \land$
$N' - 1 = N'' \land F' = N'' \times F'' \land N'' \geq 1$

$$\begin{tikzcd}[column sep=small]
 & \langle fac(2,X) \mid C_0\rangle \arrow[dl, dash, "R1"'] \arrow[d, dash, "R2"] \\
\langle 2=0, X=1 \mid C_0\rangle \arrow[d, dash] & \langle 2=N, X=N\times F, N\geq 1, fac(N-1,F) \mid C_0\rangle \arrow[d, dash] \\
\langle \blacksquare \mid C_1\rangle & \langle X=N\times F, N\geq 1, fac(N-1,F) \mid C_2\rangle \arrow[d, dash] \\
 & \langle N\geq 1, fac(N-1,F) \mid C_3\rangle \arrow[d, dash] \\
 & \langle fac(N-1,F) \mid C_4\rangle \arrow[dl, dash, "R1"'] \arrow[d, dash, "R2"] \\
\langle N-1=0, F=1 \mid C_4\rangle \arrow[d, dash] & \langle N-1=N', F=N'\times F', N'\geq 1, fac(N'-1,F') \mid C_4\rangle \arrow[d, dash] \\
\langle \blacksquare \mid C_5\rangle & \langle F=N'\times F', N'\geq 1, fac(N'-1,F') \mid C_6\rangle \arrow[d, dash] \\
 & \langle N'\geq 1, fac(N'-1,F') \mid C_7\rangle \arrow[d, dash] \\
 & \langle fac(N'-1,F') \mid C_8\rangle \arrow[dl, dash, "R1"'] \arrow[d, dash, "R2"] \\
\langle N'-1=0, F'=1 \mid C_8\rangle \arrow[d, dash] & \langle N'-1=N'', F'=N''\times F'', N''\geq 1, fac(N''-1,F'') \mid C_8\rangle \arrow[d, dash] \\
\langle F'=1 \mid C_9\rangle \arrow[d, dash] & \langle F'=N''\times F'', N''\geq 1, fac(N''-1,F'') \mid C_{11}\rangle \arrow[d, dash] \\
\langle \square \mid C_{10}\rangle & \langle N''\geq 1, fac(N''-1,F'') \mid C_{12}\rangle \arrow[d, dash] \\
 & \langle \blacksquare \mid C_{13}\rangle
\end{tikzcd}$$

Figure 4.4 Derivation tree for `fac(2,X)`.

Examination of the derivation tree shows that the goal `fac(2,X)` has four derivations, one for each leaf node in the tree. Three are failed while one is the successful derivation shown in Figure 4.3.

It may be the case that all of the derivations for a particular goal are failed, in which case there are no answers to the goal.

$$\begin{tikzcd}
 & \langle fac(0,2) \mid true \rangle \arrow[dl, dash, "R1"'] \arrow[d, dash, "R2"] \\
\langle 0 = 0, 2 = 1 \mid true \rangle \arrow[d, dash] & \langle 0 = N, 2 = N \times F, N \geq 1, fac(N-1,F) \mid true \rangle \arrow[d, dash] \\
\langle 2 = 1 \mid 0 = 0 \rangle \arrow[d, dash] & \langle 2 = N \times F, N \geq 1, fac(N-1,F) \mid 0 = N \rangle \arrow[d, dash] \\
\langle \blacksquare \mid 0 = 0 \wedge 2 = 1 \rangle & \langle N \geq 1, fac(N-1,F) \mid 0 = N \wedge 2 = N \times F \rangle \arrow[d, dash] \\
 & \langle \blacksquare \mid 0 = N \wedge 2 = N \times F \wedge N \geq 1 \rangle
\end{tikzcd}$$

Figure 4.5 Derivation tree for `fac(0,2)`.

Definition 4.7
If a state or goal G has a finite derivation tree and all derivations in the tree are failed, G is said to *finitely fail.*

The derivation tree for the goal `fac(0,2)` is shown in Figure 4.5. Both derivations contained in the tree are failed so the goal has finitely failed.

A derivation tree is not necessarily finite. Infinite derivations may occur when the program is recursive. Consider the simple program:

```
stupid(X) :- stupid(X).            (S1)
stupid(1).                         (S2)
```

The top part of the derivation tree for goal `stupid(X)` is shown in Figure 4.6. The leftmost branch is infinite.

4.5 Goal Evaluation

CLP systems evaluate a goal by constructing the derivation tree for the goal and searching through it for successful derivations. Indeed, because evaluation is essentially a search through the derivation tree, the derivation tree is sometimes called the *search space*.

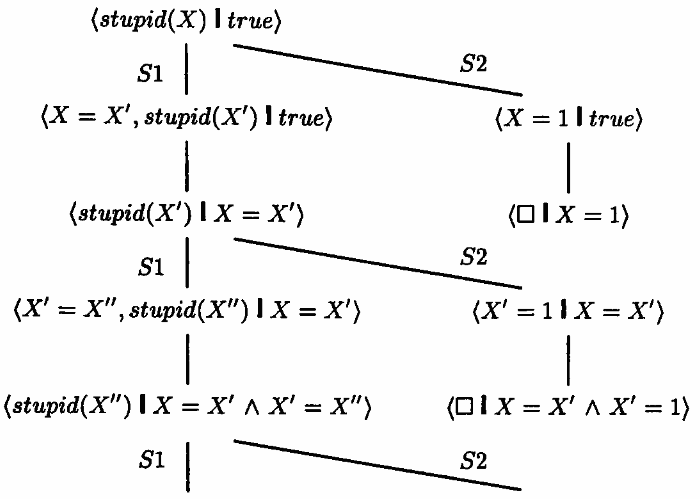

Figure 4.6 Derivation tree for `stupid(X)`.

Definition 4.8
A goal is *evaluated* by performing an in-order, that is depth-first left-to-right, traversal of the goal's derivation tree. Whenever a success state is encountered the system returns the corresponding answer to the user. The user is given the option either to halt the execution or to continue traversing the tree to find more answers. Execution halts either when the user stops the search for more answers or when the entire tree has been explored.

It is important to realise that the CLP system does not construct the entire derivation tree before traversing it to look for success states. This would be very inefficient, especially if only the first answer is required. Instead the CLP system constructs the tree as it is traversed. Exactly how this is done is discussed more fully in Chapter 10.

For example, the goal `fac(2,X)` is evaluated as follows. Essentially, the CLP system searches the derivation tree shown in Figure 4.4. Evaluation begins at the root of the tree $\langle fac(2, X) \mid C_0\rangle$. This is a choicepoint as there is more than one rule in the definition of `fac`. The first rule in the definition, $R1$, is tried first. This leads to an exploration of the left branch in the tree. Using this rule, a derivation step is applied to $\langle fac(2, X) \mid C_0\rangle$ giving $\langle 2 = 0, X = 1 \mid C_0\rangle$. A further derivation step is applied giving the child $\langle \blacksquare \mid C_1\rangle$. This is a fail state since the constraint C_1 is unsatisfiable. Evaluation, therefore, *backtracks* up the derivation tree to the closest state with unexplored children. In this case it is the root $\langle fac(2, X) \mid C_0\rangle$. Now the system tries the second rule in the definition of `fac`, effectively exploring the right branch in the tree. Rewriting with this rule gives the state $\langle 2 = N, X = N \times F, N \geq 1, fac(N - 1, F) \mid C_0\rangle$. Application of subsequent derivation steps produce the states

$\langle X = N \times F, N \geq 1, fac(N-1, F) \mid C_2 \rangle$,
$\langle N \geq 1, fac(N-1, F) \mid C_3 \rangle$ and
$\langle fac(N-1, F) \mid C_4 \rangle$.

This state is another choicepoint. Evaluation will try the first rule, $R1$, first, leading to an exploration of the left branch. Similarly to above, this leads to a failed state so execution backtracks to this state. Now the second rule is tried, leading to an exploration of the right branch. Derivation steps produce the states

$\langle N-1 = N', F = N' \times F', N' \geq 1, fac(N'-1, F') \mid C_4 \rangle$,
$\langle F = N' \times F', N' \geq 1, fac(N'-1, F') \mid C_6 \rangle$,
$\langle N' \geq 1, fac(N'-1, F') \mid C_7 \rangle$ and
$\langle fac(N'-1, F') \mid C_8 \rangle$

which are visited in turn.

Again evaluation has reached a choicepoint and again the first rule, $R1$, is tried. This leads to the exploration of the left branch which contains the following states $\langle N'-1 = 0, F' = 1 \mid C_8 \rangle$, $\langle F' = 1 \mid C_9 \rangle$ and finally $\langle \Box \mid C_{10} \rangle$. Since $solv(C_{10})$ is *true* a success state has been reached. This reflects the fact that the path from the root of the tree to this state is a successful derivation. The CLP system returns the answer $simpl(C_{10}, \{X\})$, which is $X = 2$.

Since the derivation tree has not been fully explored, the system asks the user whether they wish to continue the search. If the user answers yes, the search continues, otherwise execution of the goal finishes. In this case, if the user asks for more answers the search backtracks up the tree to the first state with a remaining unexplored child. This is $\langle fac(N'-1, F') \mid C_8 \rangle$. The second rule, $R2$, for `fac` is now tried, leading to an exploration of the right branch from this state. In turn, the states

$\langle N'-1 = N'', F' = N'' \times F'', N'' \geq 1, fac(N''-1, F'') \mid C_8 \rangle$,
$\langle F' = N'' \times F'', N'' \geq 1, fac(N''-1, F'') \mid C_{11} \rangle$,
$\langle N'' \geq 1, fac(N''-1, F'') \mid C_{12} \rangle$ and
$\langle \blacksquare \mid C_{13} \rangle$

are created and visited. Since a failed state has been reached, the system attempts to backtrack. However, there are no states with unexplored children, so the system halts with the message that there are no more answers since the entire derivation tree has now been explored.

When evaluating a goal which is finitely failed a CLP system will return the answer *no* indicating that the goal has no answers. As an example, the derivation tree for the goal `fac(0,2)` is shown in Figure 4.5. Evaluation proceeds by visiting the state $\langle fac(0,2) \mid true \rangle$ and proceeding down the left branch to states $\langle 0 = 0, 2 = 1 \mid true \rangle$, $\langle 2 = 1 \mid 0 = 0 \rangle$ and $\langle \blacksquare \mid 0 = 0 \land 2 = 1 \rangle$. Evaluation now backtracks to state $\langle fac(0,2) \mid true \rangle$ and tries the right branch. The right branch also leads to a failed state, so, since the entire tree has been explored without finding any answers, the answer *no* is returned.

For goals with infinite derivation trees, evaluation may not terminate. Evaluation of the goal `stupid(X)` whose derivation tree is illustrated in Figure 4.6 never returns an answer, even though the goal has successful derivations. This is because the CLP system spends all of its time searching down the infinite leftmost branch and never finds a successful derivation.

We will return to the issue of non-termination and how to overcome it in Chapter 7. In this case it is simple to remove the problem by reversing the order of the two rules defining predicate `stupid` so that the fact comes first. The reordered program is

```
stupid(1).
stupid(X) :- stupid(X).
```

Evaluation of the goal with this program corresponds to searching the derivation tree in Figure 4.6 from right-to-left rather than left-to-right. First the state $\langle stupid(X) \mid true\rangle$ is visited, then $\langle X = 1 \mid true\rangle$, and then $\langle \square \mid X = 1\rangle$. The answer $X = 1$ is returned to the user. If the user asks for evaluation to continue, it backtracks up to state $\langle stupid(X) \mid true\rangle$, then visits $\langle X = X', stupid(X') \mid true\rangle$, $\langle stupid(X') \mid X = X'\rangle$, $\langle X' = 1 \mid X = X'\rangle$ and then $\langle \square \mid X = X' \wedge X' = 1\rangle$. The second answer, which is again $X = 1$, is returned to the user. This example illustrates how the user can obtain more than one answer to a goal. In fact this goal has an infinite number of answers all of which are equivalent.

4.6 Simplified Derivation Trees

Unfortunately, except for toy programs, derivation trees are too large and cumbersome to illustrate evaluation of a goal. Here we introduce a variant—"simplified derivation trees"—which provide a more compact representation for a derivation tree.

One key insight behind simplified derivation trees is that only the variables from the original goal and those appearing in the goal part of the state can be of interest in the remaining derivation. Therefore, the constraint can be simplified with respect to these variables. The second insight is that derivation steps in which a primitive constraint is successfully added to the constraint store are straightforward. The steps of real interest in a derivation and derivation tree occur when user-defined constraints are rewritten; in particular, we are interested in which rule in the definition is used for the rewriting.

Definition 4.9

A state $\langle G_0 \mid C_0\rangle$ occurring in a derivation or derivation tree for goal G can be simplified in the following way. Let $V = vars(G)$ and $V_0 = vars(G_0)$. First, we compute $simpl(C_0, V \cup V_0)$. Call the result C_1. Second, for each variable $x \in V_0 - V$ which appears exactly once in C_1 in an equation of the form $x = t$, we replace

x everywhere in G_0 by t, obtaining the goal G_2. Finally, we simplify again in order to remove the substituted variables. That is to say, we compute C_2 as $simpl(C_1, V \cup vars(G_2))$. The ***simplified state*** is $\langle G_2 \mid C_2 \rangle$.

Consider the state $\langle fac(N' - 1, F') \mid C_0 \rangle$ where C_0 is

$$2 = N \wedge X = N \times F \wedge N \geq 1 \wedge N - 1 = N' \wedge F = N' \times F' \wedge N' \geq 1$$

which occurs in a derivation tree for the goal `fac(2, X)`. First, C_0 is simplified with respect to the variables $\{X, N', F'\}$ giving C_1:

$$N' = 1 \;\wedge\; X = 2 \times F'.$$

Then, G_2 is constructed by replacing N' by 1. This, if we take some liberty in simplifying expressions, gives $fac(0, F')$. Finally, C_2 is computed to be $simpl(C_1, \{X, F'\})$, which is $X = 2 \times F'$. Thus the simplified state is

$$\langle fac(0, F') \mid X = 2 \times F' \rangle.$$

Definition 4.10
A state in a derivation is *critical* if

- it is the first or last state; or
- the first literal in the state is a user-defined constraint.

A ***simplified derivation*** for goal G is obtained from a derivation for G by removing states which are not critical from the derivation and simplifying the remaining states in the derivation.
A ***simplified derivation tree*** for G is obtained by simplifying each of the derivations that form it.

For example, the simplified form of the derivation shown in Figure 4.3 is:

$$\begin{array}{c} \langle fac(2, X) \mid true \rangle \\ \Downarrow R2 \\ \langle fac(1, F) \mid X = 2 \times F \rangle \\ \Downarrow R2 \\ \langle fac(0, F') \mid X = 2 \times F' \rangle \\ \Downarrow R1 \\ \langle \square \mid X = 2 \rangle \end{array}$$

The simplified form of a derivation is much smaller than the original derivation, yet essentially contains the same information. However, some dependencies between variables in different states can be lost. For instance, the connection between the variables F and F' in the above simplified derivation is not visible.

$$\begin{tikzcd}
 & \langle fac(2,X) \mid true \rangle \arrow[dl, dash, "R1"'] \arrow[d, dash, "R2"] \\
\langle \blacksquare \mid false \rangle & \langle fac(1,F) \mid X = 2 \times F \rangle \arrow[dl, dash, "R1"'] \arrow[d, dash, "R2"] \\
\langle \blacksquare \mid false \rangle & \langle fac(0,F') \mid X = 2 \times F' \rangle \arrow[dl, dash, "R1"'] \arrow[d, dash, "R2"] \\
\langle \square \mid X = 2 \rangle & \langle \blacksquare \mid false \rangle
\end{tikzcd}$$

Figure 4.7 Simplified derivation tree for `fac(2,X)`.

The simplified form of the derivation tree shown in Figure 4.4 is shown in Figure 4.7. Note that in a simplified derivation tree the answer constraints for the original goal appear in the successful leaves of the derivation and failed derivation end in a state whose constraint is *false*.

4.7 The CLP Scheme

Although all the examples we have examined so far have used arithmetic constraints, the definitions for rules, derivations and evaluation do not depend on any particular constraint domain. Rather, they provide a scheme for creating languages which share the same evaluation mechanism.

This ***constraint logic programming scheme*** defines a family of languages. Given a particular constraint domain, constraint solver and constraint simplifier, the scheme defines a language for writing programs and a mechanism for evaluating goals and programs written in this language. The constraint domain details the primitive constraints in the language, while the solver and simplifier are used in the evaluation of goals.

For example, the tree constraint domain, *Tree*, together with the tree solver, tree_solve, and simplifier, tree_simplify, give rise to a constraint logic programming language, which we call *CLP*(*Tree*). The following is an example of a *CLP*(*Tree*) program. It defines the predicate `delete(X,Y,Z)` which deletes an element Y from list X to give list Z.

```
delete(cons(Y,X), Y, X).
delete(cons(A,X), Y, cons(A,Z)) :- delete(X, Y, Z).
```

Our reasons for requiring constraint solvers to be well-behaved become apparent in the context of the CLP scheme. The most important property is that the solver is variable name independent. This means that the behaviour of the solver, and so of the CLP system, is the same, regardless of which renamings are used in the derivation steps. The other properties ensure that goals which contain the same literals, but possibly in a different order, have the same answers. We investigate this further in the next section.

The importance of constraint simplification is illustrated by the size and complexity of the final constraint store for even simple goals. For instance, compare the final constraint store for the successful derivation for `fac(2,X)`

$$2 = N \land X = N \times F \land N \geq 1 \land N - 1 = N' \land F = N' \times F' \land N' \geq 1 \land$$
$$N' - 1 = 0 \land F' = 1$$

with the result simplified on to X, namely $X = 2$. Being able to simplify the answer with respect to the variables in the original goal by using a projecting or weakly projecting simplifier greatly improves intelligibility of the result, since many intermediate variables may be introduced in a derivation.

In the remainder of this book we shall make use of three CLP systems. The first is *CLP*(*Tree*), the constraint logic programming language over tree constraints introduced above. This is, essentially, the pure component of the logic programming language, Prolog. Tree constraints are present in almost all CLP languages because they allow the programmer to build and access data structures—a necessity for modelling of more complex problems. We shall investigate the use of tree constraints for data structure manipulation in Chapter 6.

The second language is $CLP(\mathcal{R})$. This language results from combining the *Real* and *Tree* constraint domains in a straightforward and natural manner. Elements of the domain are trees (as for the *Tree* domain) with the addition that any arithmetic constant is also a tree (with no children). Thus, domain elements are trees which may have arithmetic constants as leaves. Real constants are available using the usual decimal representation and the usual arithmetic functions are provided for reals: +, -, * and /; as well as trigonometric functions, exponentiation, minimization and maximization : `sin`, `cos`, `pow`, `min`, and `max`. Primitive constraints for arithmetic terms are equality (=), inequality ($\leq$, $\geq$), and strict inequality (<, >). We will not make use of arithmetic disequality ($\neq$) although it is available in some implementations. For trees there are only two primitive constraints: equality (=) and disequality ($\neq$) between terms. Disequality is not available in all CLP systems, but later, in Chapter 9, we shall discuss how it can be implemented on top of a solver providing only equality.

The third language, *CLP*(*FD*), combines tree constraints with finite domain constraints. Elements of the constraint domain are trees which may have integer constants as leaves. Finite domain variables either range over a finite domain of integers or a finite domain of arbitrary constants which are treated as names for integers. The usual mathematical constraints are provided for integer finite domain variables, that is equality (=), inequality ($\leq$, $\geq$)), strict inequality (<, >) and disequality ($\neq$). We shall also use complex constraints such as `element` and `cumulative`. The constraints over trees are the same as for $CLP(\mathcal{R})$, namely equality (=) and disequality ($\neq$). This means that complex data structures can be created and manipulated in the same way as in $CLP(Tree)$ and $CLP(\mathcal{R})$.

4.8 (*) Independence from Rule Ordering and Literal Selection

The answers for a goal are obtained from the successful derivations for a goal. One of the nice properties of constraint logic programs is that we can modify a program in a number of ways without altering the answers to a goal. This allows us to modify the program to improve its efficiency of execution without changing the answers—a subject we shall investigate in Chapter 7. In this section we demonstrate that the answers to a goal are independent of the order of rules in a program and the order of the literals in a goal or rule body.

It is easy to see that the answers to a goal are not effected by the order of the rules defining a predicate p. The only effect the ordering has is on the order of the children of a state in which p is rewritten. Clearly, the same set of derivations exist regardless of the order, although they may appear in different places in the tree.

However, although rule ordering does not change the answers themselves it may change the order in which the answers will be discovered or, indeed, if they will ever be discovered. This is illustrated by the `stupid` predicate. For the rules given on page 146, evaluation of the goal `stupid(X)` will fail to find any answers even though they exist in the derivation tree. However, if the rules are reordered as shown on page 149, evaluation will find an infinite number of answers.

Now let us consider the effect of the order in which literals are processed. In a derivation the literals in the goal are treated left-to-right. That is, the leftmost literal in the goal is either added to the constraint store, if it is a primitive constraint, or rewritten, if it is a user-defined constraint. We shall now show that the same answers are obtained even if the literals are processed in a different order. This means that a goal and program will give the same answers independent of the ordering of literals in the goal and in the rule bodies. This property is important because it means the CLP programmer can improve efficiency by reordering goals or bodies in a correct program with the knowledge that the answers will not change.

To make this discussion more precise, we first need to extend our notion of derivation so that we do not, necessarily, process the first literal in the goal. Instead, we allow any literal from the goal to be *selected* for processing.

Definition 4.11
A *selection derivation step* from $\langle G_1 \mid C_1 \rangle$ to $\langle G_2 \mid C_2 \rangle$, written $\langle G_1 \mid C_1 \rangle \Rightarrow \langle G_2 \mid C_2 \rangle$, is defined as follows. Let G_1 be the sequence of literals

$$L_1, \ldots, L_{i-1}, L_i, L_{i+1}, \ldots, L_m$$

and let L_i be the *selected literal.* There are two cases.

1. L_i is a primitive constraint. Then C_2 is $C_1 \wedge L_i$ and, if $solv(C_2) \equiv false$, G_2 is the empty goal; otherwise G_2 is $L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m$.
2. L_i is a user-defined constraint. Then C_2 is C_1 and G_2 is a rewriting of G_1 at L_i by some rule R in the program using a renaming ρ such that the variables in $\rho(R)$ are different from those in C_1 and G_1. If there is no rule defining the predicate of L_i then C_2 is $false$ and G_2 is the empty goal.

A *selection derivation* for state $\langle G_0 \mid C_0 \rangle$ is a sequence of states

$$\langle G_0 \mid C_0 \rangle \Rightarrow \langle G_1 \mid C_1 \rangle \Rightarrow \langle G_2 \mid C_2 \rangle \Rightarrow \ldots$$

such that for each $i \geq 0$ there is a selection derivation step from $\langle G_i \mid C_i \rangle$ to $\langle G_{i+1} \mid C_{i+1} \rangle$ and the renamed rules used in each selection derivation step do not contain variables which occur earlier in the derivation.

Derivations are a particular case of selection derivations in which the leftmost literal in the goal is always selected. However, other strategies for literal selection are possible. For instance, a derivation from the goal `fac(2,X)` in which the rightmost literal is selected is shown in Figure 4.8. The selected literal is underlined in each state.

Somewhat surprisingly, answers are independent of literal selection. To see why, consider Figure 4.8 again. The corresponding derivation in which the leftmost literal in the goal is always selected is shown in Figure 4.3. Both derivations are successful and give rise to the same answer $X = 2$. The point is that, regardless of the order in which literals are selected, if a derivation is successful then all literals will be selected. The only difference is the order of the primitive constraints in the constraint store. However, because the constraint solver is well-behaved, this cannot change the behaviour of the CLP system.

Independence from literal selection strategy depends on the constraint solver being well-behaved since, as we have seen, different literal selection strategies can lead to different orders of primitive constraints in the constraint store. If the solver is not set based, the solver might answer $false$ for one constraint and *unknown* for the reordered constraint. One derivation will fail and the other succeed, and so the answers will be different.

$$\begin{gathered}
\langle \underline{fac(2,X)} \mid true \rangle \\
\Downarrow\ R2 \\
\langle 2 = N, X = N \times F, N \geq 1, \underline{fac(N-1,F)} \mid true \rangle \\
\Downarrow\ R2 \\
\langle 2 = N, X = N \times F, N \geq 1, N-1 = N', F = N' \times F', N' \geq 1, \underline{fac(N'-1,F')} \mid true \rangle \\
\Downarrow\ R1 \\
\langle 2 = N, X = N \times F, N \geq 1, N-1 = N', F = N' \times F', N' \geq 1, N'-1 = 0, \underline{F' = 1} \mid true \rangle \\
\Downarrow \\
\langle 2 = N, X = N \times F, N \geq 1, N-1 = N', F = N' \times F', N' \geq 1, \underline{N'-1 = 0} \mid F' = 1 \rangle \\
\Downarrow \\
\langle 2 = N, X = N \times F, N \geq 1, N-1 = N', F = N' \times F', \underline{N' \geq 1} \mid F' = 1 \wedge N'-1 = 0 \rangle \\
\Downarrow \\
\langle 2 = N, X = N \times F, N \geq 1, N-1 = N', \underline{F = N' \times F'} \mid F' = 1 \wedge N'-1 = 0 \wedge N' \geq 1 \rangle \\
\Downarrow \\
\langle 2 = N, X = N \times F, N \geq 1, \underline{N-1 = N'} \mid F' = 1 \wedge N'-1 = 0 \wedge N' \geq 1 \wedge F = N' \times F' \rangle \\
\Downarrow \\
\langle 2 = N, X = N \times F, \underline{N \geq 1} \mid F' = 1 \wedge N'-1 = 0 \wedge N' \geq 1 \wedge F = N' \times F' \wedge N-1 = N' \rangle \\
\Downarrow \\
\langle 2 = N, \underline{X = N \times F} \mid F' = 1 \wedge N'-1 = 0 \wedge N' \geq 1 \wedge F = N' \times F' \wedge N-1 = N' \wedge N \geq 1 \rangle \\
\Downarrow \\
\langle \underline{2 = N} \mid F' = 1 \wedge N'-1 = 0 \wedge N' \geq 1 \wedge F = N' \times F' \wedge N-1 = N' \wedge \\
N \geq 1 \wedge X = N \times F \rangle \\
\Downarrow \\
\langle \Box \mid F' = 1 \wedge N'-1 = 0 \wedge N' \geq 1 \wedge F = N' \times F' \wedge N-1 = N' \wedge N \geq 1 \wedge \\
X = N \times F \wedge 2 = N \rangle
\end{gathered}$$

Figure 4.8 A right-to-left derivation for `fac(2,X)`.

For example, imagine

$$solv(F \leq 0 \wedge 0 = 0 \wedge F = 1) = unknown$$

but

$$solv(0 = 0 \wedge F = 1 \wedge F \leq 0) = false.$$

Then the two derivations for the goal

```
F ≤ 0, fac(0,F)
```

illustrated below, using different literal selections give different answers.

$$\begin{array}{cc}
\langle \underline{F \leq 0}, fac(0,F) \mid true \rangle & \langle F \leq 0, \underline{fac(0,F)} \mid true \rangle \\
\Downarrow & \Downarrow \; R1 \\
\langle \underline{fac(0,F)} \mid F \leq 0 \rangle & \langle F \leq 0, \underline{0 = 0}, F = 1 \mid true \rangle \\
\Downarrow \; R1 & \Downarrow \\
\langle \underline{0 = 0}, F = 1 \mid F \leq 0 \rangle & \langle F \leq 0, \underline{F = 1} \mid 0 = 0 \rangle \\
\Downarrow & \Downarrow \\
\langle \underline{F = 1} \mid F \leq 0 \wedge 0 = 0 \rangle & \langle \underline{F \leq 0} \mid 0 = 0 \wedge F = 1 \rangle \\
\Downarrow & \Downarrow \\
\langle \Box \mid F \leq 0 \wedge 0 = 0 \wedge F = 1 \rangle & \langle \blacksquare \mid 0 = 0 \wedge F = 1 \wedge F \leq 0 \rangle
\end{array}$$

Similarly, if the solver is not monotonic, different answers can result, even if the solver is set based. Imagine

$$solv(F \leq 0 \wedge F = 1) = false$$

but

$$solv(F \leq 0 \wedge F = 1 \wedge 0 = 0) = unknown.$$

Therefore, since the solver is set based,

$$solv(F \leq 0 \wedge 0 = 0 \wedge F = 1) = unknown.$$

Then the two derivations for the goal

```
F ≤ 0, fac(0,F)
```

illustrated below, using different literal selections give different answers.

$$\begin{array}{cc}
\langle \underline{F \leq 0}, fac(0,F) \mid true \rangle & \langle \underline{F \leq 0}, fac(0,F) \mid true \rangle \\
\Downarrow & \Downarrow \\
\langle \underline{fac(0,F)} \mid F \leq 0 \rangle & \langle \underline{fac(0,F)} \mid F \leq 0 \rangle \\
\Downarrow \; R1 & \Downarrow \; R1 \\
\langle \underline{0 = 0}, F = 1 \mid F \leq 0 \rangle & \langle 0 = 0, \underline{F = 1} \mid F \leq 0 \rangle \\
\Downarrow & \Downarrow \\
\langle \underline{F = 1} \mid F \leq 0 \wedge 0 = 0 \rangle & \langle \blacksquare \mid F \leq 0 \wedge F = 1 \rangle \\
\Downarrow & \\
\langle \Box \mid F \leq 0 \wedge 0 = 0 \wedge F = 1 \rangle &
\end{array}$$

Independence from literal selection is one of the most important properties of constraint logic programs. It means that the program can be understood independently of the underlying control flow used in the implementation. One reason this is important is if the CLP system employs *dynamic scheduling* to choose the literal. Dynamic scheduling is provided by many CLP languages. It allows the programmer to specify run-time conditions under which the leftmost literal should not be selected. This is discussed in Chapter 9. Independence from literal selection allows the programmer to ignore the precise details about the order in which literals will be evaluated with dynamic scheduling since the same answers will result.

Given that the answers to a goal are independent from literal selection, it is natural to ask if finite failure is also independent from the literal selection.

Example 4.6
Consider the program

```
p :- p.
```

and the goal `p, 1 = 2`. With left-to-right literal selection this goal has a single infinite derivation, in which p is repeatedly rewritten to itself. With right-to-left literal selection, however, the goal has a single failed derivation, so the goal finitely fails. The derivation trees are shown below.

$$\begin{tikzcd}
\langle \underline{p}, 1 = 2 \mid true \rangle \arrow[d, dash] & \langle p, \underline{1 = 2} \mid true \rangle \arrow[d, dash] \\
\langle \underline{p}, 1 = 2 \mid true \rangle \arrow[d, dash] & \langle \blacksquare \mid 1 = 2 \rangle \\
\langle \underline{p}, 1 = 2 \mid true \rangle \arrow[d, dash] & \\
\vdots &
\end{tikzcd}$$

This example clearly shows that finite failure is not independent of literal selection. Another, more realistic example is the goal `fac(2,3)`. It is easy to verify that this goal is finitely failed with left-to-right literal selection but not for right-to-left.

The reason independence does not hold for finite failure is that in an infinite derivation a literal which will cause failure may never be selected. We can overcome this problem if we require that literal selection is *fair* in the sense that every literal occurring in an infinite derivation is eventually selected. So long as selection is fair, finite failure is independent of literal selection.

Unfortunately, always selecting the leftmost literal is not fair. For example, in the left derivation above, the literal $1 = 2$ is never selected. Therefore, since most CLP systems do not use a fair literal selection strategy, the programmer has to consider the order of literals when reasoning about finite failure.

4.9 Summary

In this chapter we have introduced the constraint logic programming scheme. This provides a generic way of creating a constraint logic programming language given a constraint domain and a solver and simplifier for that domain. We have seen that constraint logic programs are made up of rules. These let the programmer define predicates which are simply user-defined constraints or relations. Multiple rules allow a predicate to have more than one possible definition. This allows the programmer to encode relations defined by cases or relations in which there is a choice of possible definitions. Rules may also be recursive, which provides the power of a true programming language.

Given a constraint logic program, a goal can be asked. Evaluation of the goal is via a derivation. A derivation collects primitive constraints into a constraint store and rewrites user-defined constraints using rules. The derivation stops when either the constraint store is detected by the solver to be false or all user-defined constraints have been eliminated. In the second case, we have discovered a successful derivation and, by applying a simplifier to the constraint store, we can determine an answer to the goal.

In general, because there may be a choice of rules with which to rewrite a user-defined constraint, there may be more than one derivation and answer. The different derivations for a goal are grouped together in the goal's derivation tree. We have seen that the CLP system evaluates a goal by searching the goal's derivation tree using a depth-first left-to-right traversal. Whenever an answer is discovered, the system returns this to the user. The user can then ask for further exploration of the tree. If there are no answers to the goal and the derivation tree is finite, evaluation is said to finitely fail and the system returns "no." We also introduced a concise representation for derivations and derivation trees in which the states are simplified and non-critical states are removed.

Finally we have demonstrated that the answers of a goal and program are independent of the order of literals in the goal and rule bodies. This means that goals and bodies can be reordered without changing their answers, a property we will make use of later when we investigate how to make CLP programs more efficient.

4.10 Exercises

4.1. What does the following program compute?

```
p(0,0).
p(X,(X*X)+Y)  :-  p(X-1,Y).
```

Give the derivation tree and simplified derivation tree for the goal `p(2,Y)`.

4.2. For the constraint logic program

```
p(X, Y + X)  :-  X ≥ 1, p(X - 1, Y).
p(0, 0).

q(X, Y + X)  :-  X ≥ 1, Y ≥ 0, q(X - 1, Y).
q(0, 0).
```

(a) Show the derivation tree for the goal `p(X, 3)` to a depth where no more than four uses of the recursive rule are made on any derivation.

(b) Show the derivation tree for the goal `q(X, 3)` to depth 4 (that is, four recursive calls).

(c) Suppose the domain of the program P is now Boolean. That is to say, interpret 0 to be *false*, 1 to be *true*, $+$ to be $\vee$, $-$ to be $\oplus$ and $X \geq Y$ to be $\neg(X \rightarrow Y)$ (see Section 1.5.1). Determine the set of all answers to the goal `p(X,Y)`.

4.3. The evaluation of a goal depends on the constraint solver used. Give the derivation tree for the goal `p(X,Y)` for the program

```
p(X,Y)   :-  X = Y + 2, Y ≥ 0, q(X,Y).

q(X,Y)   :-  X ≤ 1, r(X, Y).
q(X,Y)   :-  X ≤ 3, r(X, Y).

r(2,0).
r(X,2).
```

using

(a) a complete real solver;

(b) a solver that uses gauss_jordan_solve to handle equations, but ignores the inequalities; and

(c) a solver that uses gauss_jordan_solve to handle equations but collects the inequalities and only checks an inequality is satisfiable when all of its variables have fixed values determined by gauss_jordan_solve.

4.4. Using the program on page 151, give the derivation tree for the goals

(a) `delete(cons(a,cons(b,nil)), X, R)` and

(b) `delete(cons(a,cons(b,nil)), b, L)`.

4.11 Practical Exercises

Almost all of the example programs in this chapter use only real arithmetic constraints. The *CLP*($\mathcal{R}$) system can be used to execute these programs. The syntax for *CLP*($\mathcal{R}$) programs is basically that of the example programs. To read in rules to the system, you must *consult* the files in which the rules are written. This can be done in three ways. The first is to use the *CLP*($\mathcal{R}$) library function, `consult`, which takes as a single argument the name of the file to be read in. The second is to type the list of filenames at the *CLP*($\mathcal{R}$) command line, remembering to end the line with a period. Finally, you can also consult files by including their names as command line arguments to the initial `clpr` command. By default the *CLP*($\mathcal{R}$) system will add a ".clpr" suffix to each filename.

For example, imagine that we have used a text editor to write the factorial program:

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

in the file `fac.clpr`. We can read these rules into our *CLP*($\mathcal{R}$) session by doing the following.

```
$ clpr

CLP(R) Version 1.2

1 ?- consult(fac).

** Yes
```

We could also have done the following:

```
1 ?- [fac].

** Yes
```

Or, we could have simply typed

```
$ clpr fac
```

Once we have read rules into the *CLP*($\mathcal{R}$) system, it is useful to be able to display them. The list commands `ls(P)` and `ls` can be used to do this. The command `ls` lists all of the rules while `ls(P)` lists the rules defining predicate *P*.

To evaluate a goal, we just type the goal at the *CLP*($\mathcal{R}$) prompt, remembering to terminate the goal with a full stop (or period).

For instance, in the following session we read in the factorial program, list the rules in the definition of `fac` and run a simple goal.

```
1 ?- [fac].

*** Yes

2 ?- ls(fac).

fac(0, 1).
fac(N, N * F):-
     N >= 1,
     fac(N - 1, F).

** Yes

3 ?- fac(X,Y).

Y = 1
X = 0

** Retry? y

Y = 1
X = 1

** Retry? y

Y = 2
X = 2

** Retry? n
```

Note that *CLP*($\mathcal{R}$) finds one answer at a time to the goal `fac(X,Y)`. If the entire derivation tree has not yet been explored, the user is asked if they want to "retry" the goal, that is, to find another answer to the goal. Answering "y" or ";" indicates the search should continue, while answering "n" or Enter/Return indicates the search should halt.

Once we have read our rules from a file, we may discover an error in our program and modify the program in the file. To *reconsult*, that is re-read, the changed file we need to use the library function `reconsult` which takes as a single argument the file to be re-read. Alternatively we can backquote the name of the file in the consult list.

For example imagine that we have modified the file `fib` and wish to reconsult the file. We can either use the *CLP*($\mathcal{R}$) command `reconsult(fac)` or the command `[‘fac]`. Much more information about using *CLP*($\mathcal{R}$) is given in the *CLP*($\mathcal{R}$) Programmers Manual included in the *CLP*($\mathcal{R}$) distribution.

SICStus Prolog can also be used to execute arithmetic constraint programs using the real arithmetic constraint solving libraries. The syntax for rules requires that all arithmetic constraints and terms occur within braces { and }. This requires considerable modification of the programs from how they are written in the text.

The factorial program above would be written in a text file (say `fac.pl`) as follows:

```
:- use_module(library(clpr)).
fac(N, F) :- {N = 0, F = 1}.
fac(N, F1)  :-  {F1 = N*F, N >= 1, N1 = N-1}, fac(N1,F).
```

The first line ensures that the constraint solver (using floating points) is loaded. The first rule is equivalent to `fac(0,1)` but since 0 and 1 are arithmetic constants they must appear between braces, so new equations must be introduced. The second line similarly introduces the new variables `F1` and `N1` to represent the expressions `N*F` and `N-1`, and all constraints are within braces.

We can read these rules into our SICStus Prolog session by doing the following. By default the suffix ".pl" is added to a filename.

```
$ sicstus

SICStus 3 #5:
| ?- consult(fac).
```

≪ lots of messages about loading ≫

```
yes
```

We could also have done the following:

```
| ?- [fac].
```

≪ lots of messages ≫

```
yes
```

The listing command `listing` lists all of the rules of the program. The command `listing(P)` lists all of the rules defining predicate symbol P.

Given the file `fac.pl` is defined as above, the following SICStus Prolog session is analogous to that given above for $CLP(\mathcal{R})$.

```
| ?- [fac].
```

≪ lots of messages about loading ≫

```
yes
```

```
| ?- listing(fac).

fac(A, B) :-
        {A=0,B=1}.
fac(A, B) :-
        {B=A*C,A>=1,D=A-1},
        fac(D, C).

yes
| ?- fac(X,Y).

X = 0.0
Y = 1.0 ? ;

X = 1.0
Y = 1.0 ? ;

X = 2.0
Y = 2.0 ? <Return/Enter>
```

In SICStus Prolog to reconsult a file we simply consult it as above. Much more information about using SICStus Prolog is given in the SICStus Prolog User's Manual.

Be warned that legitimate Prolog programs can result when the braces {} are not added around arithmetic constraints. This can result in strange behaviour. Using the file `fac.clpr` rather than `fac.pl` the following session results.

```
| ?- ['fac.clpr'].

yes
| ?- listing(fac).

fac(0, 1).
fac(A, A*B) :-
        A>=1,
        fac(A-1, B).


yes
| ?- fac(X,Y).

X = 0
Y = 1 ? ;
{INSTANTIATION ERROR: _33>=1 - arg 1}
```

The error occurs because the >= predicate has a different meaning depending on whether it occurs inside or outside the braces. Notice how the program actually managed to give the correct first answer!

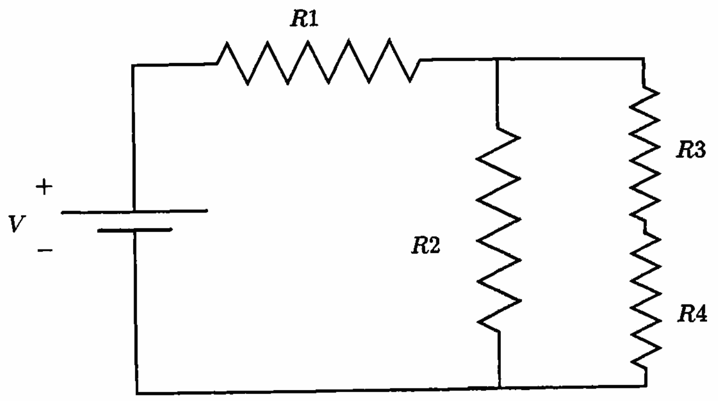

Figure 4.9 A more complex circuit to model.

P4.1. Write a rule describing the circuit shown in Figure 4.9. Use the program to find a set of available resistors and cells such that the total current is

(a) exactly 0.75 A,

(b) greater than 1.3 A.

P4.2. Write the program for computing factorial and use it to find all of the answers to the following goals:

(a) `fac(5,X).`

(b) `fac(2,3).`

(c) `fac(X,Y), X` $\leq$ `3.`

Explain the behaviour of the last goal. What would have been a better way of writing it?

P4.3. Write rules defining a predicate `abs(X,A)` which holds if A is the absolute value of X.

P4.4. Write a rule defining the user-defined constraint `line(X,Y,G,C)` which holds if the point (X, Y) lies on a line with gradient G and which intercepts the y axis at C. Use your program to:

(a) Test whether the point $(2,3)$ lies on the line $y = 4x + 2$.

(b) Find a line passing through the two points $(1,2)$ and $(2,4)$.

(c) Write a rule defining a predicate which determines the line passing between two points.

4.12 Notes

Constraint logic programming originated in the mid eighties as a combination of constraint programming and logic programming. It arose by generalising logic programming from tree constraints to other constraint domains. Jaffar *et al.* [70]

showed how the term equations of a logic programming system could be extended to equational constraints over arbitrary constraint domains, while maintaining the semantic results of logic programming. Colmerauer designed the first true constraint logic programming language PROLOG II [33], in which constraints were not only equations but, rather, were equations and *disequations* over rational trees. Jaffar and Stuckey [76] showed how the usual semantic results of logic programming could be obtained in the case of PROLOG II. Then Jaffar and Lassez [69] recognised that all these extensions were, in fact, *instances* of a more general scheme, the *constraint logic programming* scheme. It is from this paper that the term "constraint logic programming" originates.

Early work on constraint logic programming systems occurred independently at three locations: Colmerauer, at the Groupe d'Intelligence Artificielle in Marseilles, extended PROLOG II to PROLOG III [34, 35] which provided constraints over trees, strings, Booleans and real linear arithmetic; Jaffar *et al.* [74, 75], at Monash University, developed the language $CLP(\mathcal{R})$ which provided constraints over trees and real arithmetic; and Dincbas *et al.* [4], at ECRC, developed an extension to Prolog, called CHIP, which handled constraints over trees, finite domains, and finite ranges of integers.

Many good introductory articles and surveys of constraint logic programming are available. Cohen [32] gives a short introduction and an historical overview. Jaffar and Maher [71] give a more in-depth survey and include a wealth of references. The paper by Jaffar *et al.* [72] provides a good introduction to the semantics of constraint logic programs. In particular, it provides proofs for the various independence results concerning literal scheduling.

A large variety of problems have been solved successfully by using constraint logic programming languages over various constraint domains. Wallace [140] gives a number of examples of industrial applications of constraint logic programming. We now briefly discuss a few example applications.

The most significant industrial impact of constraint logic programming languages has been in solving management decision problems traditionally solved by operations research methods, for example cutting stock problems [42] and scheduling problems [43]. The book by Van Hentenryck [135] shows how a wide variety of problems can be solved in constraint logic programming over finite domains.

Another major application area has been electrical circuit analysis, synthesis and diagnosis [62, 56, 95, 118, 119]. There are also examples of problem solving in civil engineering [81]. Engineering applications tend to combine hierarchical composition of complex systems, mathematical or Boolean models, and—especially in the case of diagnosis and design—deep rule-based reasoning. Because of this, constraint logic programming has proved to be well suited for these problems.

Other areas of application include options trading [82, 66] and financial planning [10, 11, 19], where solutions often take the form of expert systems involving mathematical models. More exotic applications have included restriction site mapping in genetics [146] and generating test data for communications protocols [55].