11 Constraint Databases

In this chapter we investigate another way of viewing constraint logic programs. Databases allow users to store data, such as names and addresses, and then find data which satisfies a query, for example, finding all of the people living in a particular suburb. We can also think of a constraint logic program as a kind of database, called a *constraint database*.

However, the evaluation mechanism for CLP is not well suited to database applications. Usually when querying a database, the user wishes to find *all* answers to the query at once, rather than laboriously backtracking through each answer and asking for another. For this reason, viewing constraint logic programs as constraint databases leads to a different execution strategy, so called *bottom-up* evaluation, which is closer to standard database evaluation techniques.

Constraint databases are very powerful. They extend relational, deductive and spatial databases and have several advantages over these more traditional types of databases. Throughout this book we have seen how we can use constraints to build simple models of even quite complex relations. In the same way constraints allow complex data to be modelled succinctly using constraint databases. Another advantage of constraints is that different types of data, for example, spatial, temporal and relational data, can all be manipulated using the same uniform mechanism—constraints.

11.1 Modelling with Constraint Databases

In this section we illustrate how simple CLP programs, called *constraint databases*, can be used to model applications which are traditionally handled using databases.

In Section 5.2 we saw how to model a genealogical database using CLP.

Example 11.1
The database contains two relations. The `father(X, Y)` predicate holds when X is the father of Y and similarly the `mother(X, Y)` predicate holds when X is the mother of Y. The database consists of the facts:

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

Using rules it was simple to express a variety of queries. For instance, the following rules defined the constraint `parent(X,Y)` to hold when X is a parent of Y.

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

We will now look at two rather more interesting applications. Constraint databases allow the natural expression of spatial information using arithmetic constraints to model shapes.

Example 11.2
George and Rosemary have recently purchased a potential gold mine site. It is shown on the map in Figure 11.1. The areas delineated by the continuous lines show the expected location of the gold ore seams under the ground. The circles detail the sites which are suitable for drilling shafts to reach the gold. The origin (0,0) of the map is represented by two vectors each of unit length. They wish to model their gold mine site using a constraint database and use this to determine the best place to install a mine shaft. How can they do this?

Notice that the shapes of the gold ore seams are not necessarily convex. (A shape is *convex* if a line drawn between any two points within the shape stays entirely within the shape). Non-convex objects are difficult, if not impossible, to represent with only a single constraint, so, instead of representing each gold seam by a single data item, we will model it by breaking it into convex polygons of roughly equal size. These polygons will approximately follow the seam's boundaries and can be represented using a linear arithmetic constraint which holds whenever a point lies within the polygon.

For example, the point (X, Y) is in region a of the leftmost gold seam whenever the constraint

$$X + Y \geq 2 \land X \geq Y \land X \leq 2$$

holds. Using this approach we can model the gold seams by using the predicate `ore`. It has multiple rules, one for each convex region. Each rule details the name of the region and its shape. For instance, region a is modelled by the rule

```
ore(N, X, Y) :- N = a, X + Y ≥ 2, X ≥ Y, X ≤ 2.
```

This is a good example of the utility of constraints for representing heterogeneous data—constraints are used to model both the name of the region and its shape.

Suitable drilling sites can be represented as a point together with a name. The predicate `shaft` details the possible drilling sites with a rule for each site. For

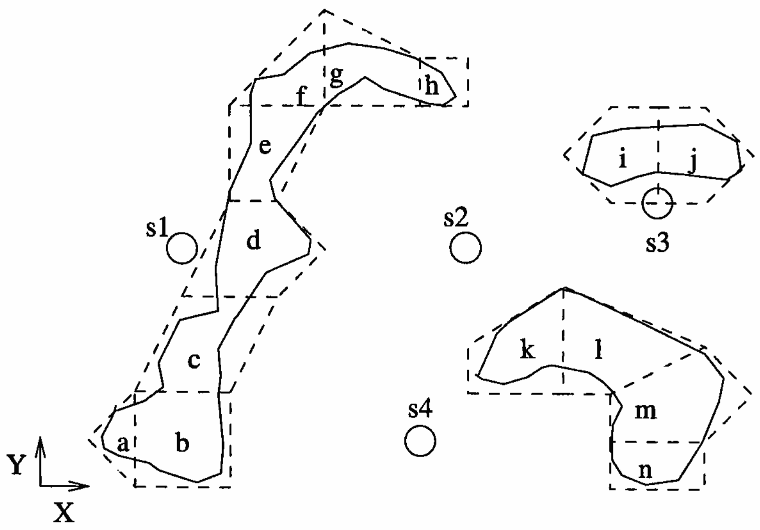

Figure 11.1 Map of a gold mine site.

instance, shaft site *s*2 is modelled by the rule

```
shaft(N, X, Y) :- N = s2, X = 9, Y = 5.
```

In order to choose where to drill the shaft we might be interested in finding the regions of ore, O, that are within a certain radius, D, of a proposed shaft site, S. We can model this query using the rule

```
within(S, D, O) :-
    (XO - XS)² + (YO - YS)² ≤ D²,
    shaft(S, XS, YS),
    ore(O, XO, YO).
```

For instance, the goal `within(s2, 4, O)` gives the answers

$$O = d, \quad O = g, \quad O = h,$$
$$O = i, \quad O = k, \quad O = l.$$

Constraint databases also allow natural modelling of temporal data. That is to say, data which changes over time. We can model time by integer values, so time $t + 1$ is the next time instant after time t (if desired, we could also model time using real numbers). Arithmetic constraints over integers allow us to reason about time very naturally. We shall reserve the first argument of every rule to specify the interval of time for which the data in the rule is valid.

Example 11.3

Consider the employment records of CLP Inc. As time progresses the positions of people change in the company—new employees arrive, old employees leave and the management structure of the company changes. We can represent this data using two predicates. The first, `job(T, Name, Posn, Dept)`, details the position *Posn* and department *Dept* of employee *Name* at time T. The second predicate, `manager_of(T, Dept, Name)`, shows the manager of department *Dept* at time T.

Rather than storing individual tuples for each different time we can use constraints to define time intervals. For example, if *bart* is the manager of the *sales* department from 1980–1992 and the manager of *marketing* from 1993–1996, while *maria* is the manager of *sales* in 1996 we can represent this by

```
manager_of(T, sales, bart) :- 1980 ≤ T, T ≤ 1992.
manager_of(T, marketing, bart) :- 1993 ≤ T, T ≤ 1996.
manager_of(T, sales, maria) :- T = 1996.
```

We will assume that the `job` relation is defined by

```
job(T, peter, salesperson, sales) :- 1985 ≤ T, T ≤ 1987.
job(T, peter, consultant, sales) :- T = 1988.
job(T, peter, analyst, support) :- 1989 ≤ T, T ≤ 1996.
job(T, kim, salesperson, sales) :- 1994 ≤ T, T ≤ 1995.
job(T, maria, salesperson, sales) :- 1988 ≤ T, T ≤ 1991.
job(T, maria, consultant, support) :- 1992 ≤ T, T ≤ 1995.
job(T, maria, manager, management) :- T = 1996.
job(T, bart, manager, management) :- 1980 ≤ T, T ≤ 1996.
```

If we were to store the same data in a traditional relational database then for each of the above facts we would require one tuple for each value that T could take. Representing the above facts would require over 50 tuples. Clearly if time is broken down into smaller periods (for example days) storing individual tuples for each time period becomes impractical. For this reason, databases which allow concise modelling of time varying data are of considerable interest.

We can use this database to answer a number of simple queries in a straightforward way. For instance, we might wish to find all of the people P who have ever been managed by manager M. This is accomplished by using the rule

```
ever_managed(M, P) :- manager_of(T, D, M), job(T, P, _, D).
```

The goal `ever_managed(M, P)` with respect to the temporal database defined above has answers:

$$M = bart \wedge P = peter \qquad \text{and} \qquad M = bart \wedge P = maria.$$

We might also be interested in finding the "long term" employees of CLP Inc. We can define long term employees as those who have worked for CLP Inc for more than 10 years. They are found using the rule

```
long_term(P) :- job(T0, P, _, _), job(T1, P, _, _), T1 - T0 ≥ 10.
```

The goal `long_term(P)` has the answers $P = peter$ and $P = bart$.

11.2 Bottom Up Evaluation

The preceding examples demonstrate that CLP programs provide a powerful mechanism for modelling many database applications. Unfortunately, the backtracking evaluation mechanism for CLP we have previously described is not well suited to this type of application. There are two main reasons. First, in most database applications we are interested in finding *all* of the answers to a query at once, rather than laboriously backtracking through each and asking for another. Second, the standard evaluation mechanism may not terminate even when there are only a finite number of answers to a query.

Example 11.4
To illustrate these issues, consider the following constraint database which describes the flights offered by a new Australian airline, Oz-air Inc. Each fact `flight(Num,From,To)` details the number of the flight *Num*, its start city *From*, and its destination *To*.

```
flight(300, melbourne, sydney).
flight(301, sydney, melbourne).
flight(500, sydney, brisbane).
flight(501, brisbane, sydney).
```

A reasonable query is: "between which cities can you fly with Oz-air Inc, possibly via other cities?" The following predicate, `connects`, attempts to answer this. It captures the reasoning that you can use Oz-air to fly between two cities if there is a direct flight between the cities or a connecting flight via another city:

```
direct_flight(From, To) :- flight(_, From, To).
connects(From, To) :- direct_flight(From, To).
connects(From, To) :- direct_flight(From, Via), connects(Via, To).
```

If we evaluate the goal `connects(melbourne,T)` using the standard CLP evaluation mechanism, we will obtain the infinite sequence of answers:

$$\begin{array}{lll} T = sydney & T = melbourne & T = brisbane \\ T = sydney & T = melbourne & T = brisbane \\ T = sydney & T = melbourne & T = brisbane \\ \vdots & \vdots & \vdots \end{array}$$

The problem is that the flight relationship has cycles which are followed in the derivation. The standard evaluation mechanism is not intelligent enough to realise that it has already "visited" that city and does a large (infinite) amount of work finding the same answers again and again.

For this reason we will give another evaluation mechanism for CLP programs which is based on relational and deductive database evaluation techniques. This mechanism works on a set of answers at a time and works from the "bottom-up."

Consider the flight database again. We can evaluate the `connects(melbourne,T)` by first finding all of the answers to the goal `connects(F,T)` and then selecting those answers for which $F = melbourne$.

A rather natural way to find all of the answers to `connects(F,T)` is as follows. We first find all of the answers to the goal `flight(N,F,T)`. These are

$$\begin{aligned}&N = 300 \land F = melbourne \land T = sydney,\\&N = 301 \land F = sydney \land T = melbourne,\\&N = 500 \land F = sydney \land T = brisbane,\\&N = 501 \land F = brisbane \land T = sydney.\end{aligned}$$

They can be more conveniently represented by the set of facts, S_1,

```
flight(300, melbourne, sydney),   flight(301, sydney, melbourne),
flight(500, sydney, brisbane),    flight(501, brisbane, sydney).
```

Using these we can now find all of the answers to the goal `direct_flight(F, T)`. In essence, we use the program containing the rule defining `direct_flight` together with the above facts. This gives rise to the answers

$$\begin{aligned}&F = melbourne \land T = sydney,\\&F = sydney \land T = melbourne,\\&F = sydney \land T = brisbane,\\&F = brisbane \land T = sydney.\end{aligned}$$

Again we can conveniently represent these as a set of facts, say S_2:

```
direct_flight(melbourne, sydney),   direct_flight(sydney, melbourne),
direct_flight(sydney, brisbane),    direct_flight(brisbane, sydney).
```

Now we are in a position to compute all of the answers to `connects(F,T)`. This is an iterative process. We start by evaluating the goal `connects(F,T)` with respect to the program containing the facts in S_1 and S_2 and the first rule defining `connects`. This gives the answers,

$$\begin{aligned}&F = melbourne \land T = sydney,\\&F = sydney \land T = melbourne,\\&F = sydney \land T = brisbane,\\&F = brisbane \land T = sydney.\end{aligned}$$

We now evaluate the goal using the second rule defining `connects` and the facts in S_1 and S_2. This gives no answers since no facts match the `connects` literal at the end of the rule. We represent the answers found in this iteration by the facts, S_3,

```
connects(melbourne, sydney),   connects(sydney, melbourne),
connects(sydney, brisbane),    connects(brisbane, sydney).
```

Since the second rule for `connects` is recursive we continue this process to find more facts. We now find the answers to `connects(F,T)` using a single application of the second rule in the definition together with the facts in S_1, S_2 and S_3. This gives rise to the answers

$$\begin{aligned}
&F = melbourne \land T = melbourne,\\
&F = sydney \land T = sydney,\\
&F = brisbane \land T = brisbane,\\
&F = melbourne \land T = brisbane,\\
&F = brisbane \land T = melbourne.
\end{aligned}$$

We represent these using the facts, S_4,

```
connects(melbourne, melbourne),   connects(sydney, sydney),
connects(brisbane, brisbane),     connects(brisbane, melbourne),
connects(melbourne, brisbane).
```

We repeat the above process and find the answers to `connects(F,T)` using a single application of the second rule in the definition together with the facts in S_1, S_2, S_3 and now S_4. This gives the answers

$$\begin{aligned}
&F = melbourne \land T = sydney,\\
&F = sydney \land T = melbourne,\\
&F = sydney \land T = brisbane,\\
&F = brisbane \land T = sydney,\\
&F = melbourne \land T = melbourne,\\
&F = sydney \land T = sydney,\\
&F = brisbane \land T = brisbane,\\
&F = melbourne \land T = brisbane,\\
&F = brisbane \land T = melbourne.
\end{aligned}$$

All of these answers have been found in a previous iteration. Thus, we can stop the process since we have now found all of the answers to `connects(F,T)`.

Finally we use the facts we have generated in S_1, S_2, S_3 and S_4 to answer the original goal `connects(melbourne,T)`. This gives the answers

$$T = melbourne, \quad T = sydney, \quad T = brisbane.$$

We call this evaluation mechanism *bottom-up evaluation* and refer to the standard evaluation mechanism for CLP programs as *top-down evaluation.* The advantage of using bottom-up evaluation to answer this goal is clear: the computation finishes because it does not repeat work by cycling through the same cities again and again. We will now explore bottom-up evaluation in more detail.

At each step in a bottom-up computation, we have a set of facts for each predicate. These facts represent the answers which have been found so far for that predicate. In the above example, the facts consisted of a user-defined constraint with fixed arguments, however one of the strengths of constraint databases is that the facts can also contain constrained variables. For instance, consider the employment records of CLP Inc. discussed in Example 11.3. The answers to the goal `manager_of(T, S, P)` are represented by facts of the form

```
manager_of(T, sales, bart) :- 1980 ≤ T, T ≤ 1992,
manager_of(T, marketing, bart) :- 1993 ≤ T, T ≤ 1996,
manager_of(T, sales, maria) :- T = 1996.
```

We can extend our earlier definition of a fact to cover this case.

Definition 11.1
A *fact* is a rule of the form

$$A \text{ :- } c_1, \ldots, c_n$$

where A is a user-defined constraint and $c_1, \ldots, c_n$ are primitive constraints.

We will continue to write facts of the form A :- $\square$ simply as A.

When evaluating a program bottom-up, it is useful to simplify the facts as they are generated. This reduces the size of the facts and can also help to determine if a fact has been encountered before. Simplification of facts is closely related to constraint simplification.

But what does it mean for two facts to be equivalent? For instance, the facts

```
connects(melbourne, sydney),
connects(F, T) :- F = melbourne, T = sydney,
connects(T, F) :- T = melbourne, F = sydney,
```

are all equivalent since the names of the variables in the head of the fact do not matter—after all the fact will be renamed when it is used to answer a goal—and variables can be equated to arguments without changing the behaviour. When discussing equivalence, it will prove useful to first rewrite a fact into a "head normal form" in which the arguments in the head are distinct new variables.

Definition 11.2
The *head normal form* of the fact

$$p(t_1, \ldots, t_n) \text{ :- } C$$

is the fact

$$p(X_1^\#, \ldots, X_n^\#) \text{ :- } X_1^\# = t_1, \ldots, X_n^\# = t_n, C$$

where the variables $X_1^\#, \ldots, X_n^\#, \ldots$ are fixed distinct variables which do not appear in the body of any fact which is not head normalized.

The head normal form of the three facts

```
connects(melbourne, sydney),
connects(F, T) :- F = melbourne, T = sydney,
connects(T, F) :- T = melbourne, F = sydney
```

is

```
connects(X₁#, X₂#) :- X₁# = melbourne, X₂# = sydney,
connects(X₁#, X₂#) :- X₁# = F, X₂# = T, F = melbourne, T = sydney,
connects(X₁#, X₂#) :- X₁# = T, X₂# = F, T = melbourne, F = sydney.
```

Note that for any predicate, the head normal form of any fact defining that predicate has the same head.

By rewriting the above facts in head normal form it is now apparent that they are equivalent since the constraints in the body of each fact

$$X_1^\# = melbourne \wedge X_2^\# = sydney,$$
$$X_1^\# = F \wedge X_2^\# = T \wedge F = melbourne \wedge T = sydney,$$
$$X_1^\# = T \wedge X_2^\# = F \wedge T = melbourne \wedge F = sydney,$$

are equivalent with respect to the variables in the head of the fact, $X_1^\#$ and $X_2^\#$.

Head normal form provides a general mechanism by which we can define equivalence of facts.

Definition 11.3
Let fact F_1 have head normal form

$$A \text{ :- } c_1, \ldots, c_m$$

and fact F_2 have head normal form

$$A' \text{ :- } c'_1, \ldots, c'_n.$$

F_1 and F_2 are *equivalent* if A and A' are identical and $c_1 \wedge \cdots \wedge c_m$ is equivalent to $c'_1 \wedge \cdots \wedge c'_n$ with respect to the variables in A.

Analogously to constraint simplification, a fact simplifier takes a fact and returns a fact which is "equivalent" but, in some sense, simpler. Details of the simplification process depend on the constraint domain.

Definition 11.4
A *fact simplifier*, *fsimpl*, for constraint domain $\mathcal{D}$ takes a fact F_1 and returns a fact F_2 such that F_1 and F_2 are equivalent.

Exactly how constraints are simplified is constraint domain and implementation specific. In our examples we will use the following two rules to simplify the facts. The rules are repeatedly applied until neither is applicable.

The first rule simplifies the fact A :- $c_1, \ldots, c_n$ to

$$A \text{ :- } c'_1, \ldots, c'_m.$$

where $c'_1 \wedge \cdots \wedge c'_m$ is $simpl(c_1 \wedge \cdots \wedge c_n, vars(A))$. The two facts are equivalent since the variables in the head $vars(A)$ are the only variables of interest.

The second rule is used to simplify the fact A :- $c_1, \ldots, c_n$ when some c_i is of the form $X = t$ where X does not appear in t or elsewhere in the body. The resulting simplified fact is

$$A' \text{ :- } c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n.$$

where A' is obtained from A by replacing every occurrence of X by t.

As an example of simplification, consider the fact

connects($X_1^\#$, $X_2^\#$) :- $X_1^\#$ = F, $X_2^\#$ = T, F = melbourne, T = sydney.

Application of the first rule gives the fact

connects($X_1^\#$, $X_2^\#$) :- $X_1^\#$ = melbourne, $X_2^\#$ = sydney.

since

$$\mathsf{tree\_simplify}(X_1^\# = F \wedge X_2^\# = T \wedge F = melbourne \wedge T = sydney, \{X_1^\#, X_2^\#\})$$

is

$$X_1^\# = melbourne \wedge X_2^\# = sydney.$$

Applying the second rule twice gives the fact

`connects(melbourne, sydney).`

This is the final simplified form since now neither of the simplification rules can be applied.

The basic mechanism in bottom-up evaluation is to use the current set of facts to find the answers to a goal or to find the answers to the body of a rule.

Definition 11.5
The set of answers to a goal G *generated* from the set of facts S, written gen_ans(G, S), is the set of constraints C for which there is a successful derivation

$$\langle G \mid true \rangle \Rightarrow \cdots \Rightarrow \langle \Box \mid C \rangle$$

using the facts in S as the program.

For instance, the set of answers to the body of the second rule defining `connects`, namely

```
    direct_flight(From, Via), connects(Via, To).
```

generated from the facts in S_1, S_2 and S_3 defined above is the set

$$\begin{aligned}\{\quad & From = melbourne \land Via = sydney \land To = melbourne, \\ & From = sydney \land Via = melbourne \land To = sydney, \\ & From = sydney \land Via = brisbane \land To = sydney, \\ & From = brisbane \land Via = sydney \land To = brisbane, \\ & From = melbourne \land Via = sydney \land To = brisbane, \\ & From = brisbane \land Via = sydney \land To = melbourne \quad \}.\end{aligned}$$

We are now in a position to give the bottom-up evaluation algorithm. At each stage in the derivation there is a current set of facts S which represent the answers found so far to the program. Initially this is the empty set. We repeatedly iterate through each rule R in the program using gen_facts to find the new set of facts generated from that rule using the facts in S. After each iteration the set of new facts, S_{new}, is compared to S. If they are the same, evaluation has finished and the set of answers for the goal, generated using the final set of facts, is returned.

The call gen_facts(R, S) finds the set of answers to the body of rule R generated from the set of facts S. For each answer it builds the corresponding generated fact and then simplifies it.

The function same determines if the two sets of facts S and S_{new} contain equivalent facts. It uses new to check if there is some fact F in S_{new} which is not equivalent to any fact in S.

Algorithm 11.1: Bottom-up evaluation
INPUT: A program P and a goal G.
OUTPUT: A set of answers to G.
METHOD: The algorithm is shown in Figure 11.2. □

The facts in S_{new} generated from the old set of facts S are called the *immediate consequences* of S while those in the final set of facts produced in bottom-up evaluation are called the *program consequences*.

Example 11.5
Consider the following program P which defines the predicates `voltage_divider`, `cell`, `resistor`, `buildable_vd`, and `goal_vd` (the last predicate finds a buildable voltage divider cell that satisfies the goal from Section 4.2):

```
voltage_divider(V,I,R1,R2,VD,ID) :-
    V1 = I * R1, VD = I2 * R2,
    V = V1 + VD, I = I2 + ID.
cell(9).
resistor(5).
resistor(9).
```

```
P is a program;
G is a goal;
S, $S_{new}$ and $S_R$ are sets of facts;
R is a rule;
C is a constraint;
$c_1, \ldots, c_n$ are primitive constraints;
F and $F_1$ are facts.

bottom_up(P,G)
    $S_{new} := \emptyset$
    repeat
        $S := S_{new}$
        $S_{new} := \emptyset$
        for each rule R in P do
            $S_{new} := S_{new} \cup$ gen_facts(R, S)
        endfor
    until same(S,$S_{new}$)
    return gen_ans(G,S)

gen_facts(R, S)
    let R be of form H :- B
    $S_R := \emptyset$
    for each $C \in$ gen_ans(B, S) do
        let C be of form $c_1 \wedge \cdots \wedge c_n$
        $S_R := S_R \cup \{fsimpl(H :- c_1, \ldots, c_n)\}$
        endfor
    return $S_R$

same(S, $S_{new}$)
    for each $F \in S_{new}$ do
        if new(F,S) then return false endif
    endfor
    return true

new(F, S)
    for each $F_1 \in S$ do
        if F and $F_1$ are equivalent then return false endif
    endfor
    return true
```

Figure 11.2 Bottom-up evaluation of a program *P*.

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

$$\langle voltage\_divider(V, I, R1, R2, VD, ID), cell(V), resistor(R1), resistor(R2) \mid true\rangle$$
$$\Downarrow$$
$$\langle cell(V), resistor(R1), resistor(R2) \mid V = I * R1 + VD \land I = I2 + ID \land VD = I2 * R2\rangle$$
$$\Downarrow$$
$$\langle resistor(R1), resistor(R2) \mid V = I * R1 + VD \land I = I2 + ID \land VD = I2 * R2 \land V = 9\rangle$$
$$\Downarrow$$
$$\langle resistor(R2) \mid 9 = 5 * I1 + VD \land I = I2 + ID \land VD = I2 * RD \land V = 9 \land R1 = 5\rangle$$
$$\Downarrow$$
$$\langle \square \mid 9 = 5 * I + VD \land I = I2 + ID \land VD = 5 * I2 \land V = 9 \land R1 = 5 \land R2 = 5\rangle$$

Figure 11.3 Derivation for the body of the rule for `buildable_vd`.

Bottom-up evaluation of this program to solve the goal `goal_vd(V, R1, R2)` proceeds as follows. Initially S and S_{new} are set to the empty set of facts. The first four rules will generate the immediate consequences

```
voltage_divider(I*R1+VD,I2+ID,R1,R2,I2*R2,ID),
cell(9),
resistor(5),
resistor(9).
```

Since S_{new} is empty, the remaining rules do not generate any facts since they contain user-defined constraints in their body. Thus at the end of the first iteration, S_{new} is the set containing the above four facts. As S_{new} is not the same as S, iteration of the main loop continues.

In the second iteration, S is set to the old value of S_{new} and S_{new} is set to the empty set. Again evaluation of the first four rules will give rise to the facts

```
voltage_divider(I*R1+VD,I2+ID,R1,R2,I2*R2,ID),
cell(9),
resistor(5),
resistor(9).
```

This time the rule for `buildable_vd` allows us to find another immediate consequence of S. This is because, using the facts in S, there are successful derivations for the body of the rule for `buildable_vd`

```
voltage_divider(V,I,R1,R2,VD,ID), cell(V), resistor(R1), resistor(R2).
```

One successful simplified derivation is illustrated in Figure 11.3.

Simplifying the answer on to the variables $\{V, I, R1, R2, VD, ID\}$ gives

$$VD = -5 * I + 9 \land ID = 2 * I - 1.8 \land V = 9 \land R1 = 5 \land R2 = 5.$$

This derivation gives rise to the simplified fact:

```
buildable_vd(9, I, 5, 5, -5*I + 9, 2*I - 1.8).
```

The total set of simplified facts produced for `buildable_vd` is:

```
buildable_vd(9, I, 5, 5, -5*I + 9, 2*I - 1.8),
buildable_vd(9, I, 5, 9, -5*I + 9, 14/9*I - 1),
buildable_vd(9, I, 9, 5, -9*I + 9, 2.8*I - 1.8),
buildable_vd(9, I, 9, 9, -9*I + 9, 2*I - 1).
```

These correspond to the four choices of pairs of values for the resistors. The final rule in the program, that defining `goal_vd`, does not give rise to any immediate consequences since S does not contain any facts defining the predicate `buildable_vd`. Thus at the end of the iteration, S_{new} contains the facts

```
voltage_divider(I*R1+VD,I2+ID,R1,R2,I2*R2,ID),
cell(9),
resistor(5),
resistor(9),
buildable_vd(9, I, 5, 5, -5*I + 9, 2*I - 1.8),
buildable_vd(9, I, 5, 9, -5*I + 9, 14/9*I - 1),
buildable_vd(9, I, 9, 5, -9*I + 9, 2.8*I - 1.8),
buildable_vd(9, I, 9, 9, -9*I + 9, 2*I - 1).
```

Since this is not the same as the facts in S, the main loop is executed again.

In this iteration, S is set to the above facts. Execution proceeds as in the last iteration except that the rule defining `goal_vd` now gives rise to the immediate consequence:

```
goal_vd(9, 5, 9).
```

Thus, by the end of the iteration S_{new} contains the facts

```
voltage_divider(I*R1+VD,I2+ID,R1,R2,I2*R2,ID),
cell(9),
resistor(5),
resistor(9),
buildable_vd(9, I, 5, 5, -5*I + 9, 2*I - 1.8),
buildable_vd(9, I, 5, 9, -5*I + 9, 14/9*I - 1),
buildable_vd(9, I, 9, 5, -9*I + 9, 2.8*I - 1.8),
buildable_vd(9, I, 9, 9, -9*I + 9, 2*I - 1),
goal_vd(9, 5, 9).
```

Since this is not the same as the facts in S, the main loop is executed again.

This time no new immediate consequences are found, so the main loop terminates as S and S_{new} contain the same facts. Finally, `gen_ans`, is called with the final set of facts to answer the initial goal `goal_vd(V, R1, R2)`. This gives the single answer

$$V = 9 \land R1 = 5 \land R2 = 9.$$

11.3 Bottom-Up versus Top-Down

Both of the evaluation methods, bottom-up and top-down, have advantages and disadvantages. If they both terminate, they give rise to an equivalent set of answers. This was demonstrated in the preceding example when the answer found for `goal_vd(V, R1, R2)` using bottom-up evaluation was the same as that found using top-down evaluation (see Section 4.2). This equivalence also carries over to "failure." That is, if both methods terminate for a goal and one method gives no answers, then the other method will also give no answers.

For the user, the main difference between bottom-up and top-down evaluation is their termination behaviour. Bottom-up may not terminate when top-down does or vice versa.

Bottom-up evaluation is sometimes able to terminate when top-down evaluation runs forever. An example of this behaviour was shown in Example 11.4: with the flight database, the goal `connects(melbourne,T)` did not terminate using top-down evaluation. However, as we have seen, bottom-up evaluation terminates finding the three answers.

Bottom-up evaluation will terminate as long as the set of program consequences is finite. However, even if the set of program consequences is finite, top-down evaluation may still not terminate since it can find the same answer again and again.

On the other hand, because top-down evaluation is goal-directed it may terminate even when the set of program consequences is infinite. This occurs when all answers to the initial goal can be found by only exploring a finite subset of the program consequences. Thus top-down evaluation will sometimes terminate while bottom-up evaluation does not.

Example 11.6

Recall the factorial program from Section 4.2:

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

The goal `fac(2, X)` has the derivation tree shown in Figure 11.4. Thus, top-down evaluation of the goal will terminate after finding the single answer $X = 2$.

Now consider bottom-up evaluation of this goal. After the first iteration the set of facts S_{new} contains the fact

```
fac(0, 1).
```

After the second iteration it contains the facts

```
fac(0, 1), fac(1, 1).
```

After the third iteration it contains

```
fac(0, 1), fac(1, 1), fac(2, 2),
```

$$\begin{tikzcd}
 & \langle fac(2,X) \mid true \rangle \arrow[dl, "F1"'] \arrow[dr, "F2"] & \\
\langle \blacksquare \mid 0 = 2 \rangle & & \langle fac(N-1,F) \mid 2 = N \wedge X = N \times F \wedge N \geq 1 \rangle \arrow[dl, "F1"'] \arrow[dd, "F2"] \\
 & \langle \blacksquare \mid 2 = N \wedge X = N \times F \wedge N \geq 1 \wedge N - 1 = 0 \rangle & \\
 & & \langle fac(N1-1,F1) \mid 2 = N \wedge X = N \times F \wedge N \geq 1 \wedge N - 1 = N1 \wedge F = N1 \times F1 \wedge N1 \geq 1 \rangle \arrow[dl, "F1"'] \arrow[dd, "F2"] \\
 & \langle \square \mid 2 = N \wedge X = N \times F \wedge N \geq 1 \wedge N - 1 = N1 \wedge F = N1 \times F1 \wedge N1 \geq 1 \wedge N1 - 1 = 0 \wedge F1 = 1 \rangle & \\
 & & \langle \blacksquare \mid 2 = N \wedge X = N \times F \wedge N \geq 1 \wedge N - 1 = {'}N1 \wedge F = N1 \times F1 \wedge N1 \geq 1 \wedge N1 - 1 = N2 \wedge F1 = N2 \times N2 \wedge N2 \geq 1 \rangle
\end{tikzcd}$$

Figure 11.4 Derivation tree for goal `fac(2,X)`.

and after the fourth,

```
fac(0, 1), fac(1, 1), fac(2, 2), fac(3, 6).
```

Every iteration will produce another fact, so evaluation will not terminate. The reason is that the set of program consequences is infinite.

Because of the different termination behaviour, bottom-up evaluation may fail to terminate while top-down evaluation finitely fails and vice versa.

Example 11.7
Consider the factorial program of Example 11.6. The simplified derivation tree for the goal `fac(2,4)` is shown in Figure 11.5. Since all derivations in the tree are failed, the goal finitely fails. However, bottom-up evaluation of this goal will not terminate since the set of program consequences is infinite.

Example 11.8
Consider the flight database from Example 11.4. Using top-down evaluation the goal `connects(melbourne,darwin)` will not finitely fail since the derivation tree is infinite. With bottom-up evaluation, however, the goal `connects(melbourne,darwin)` will fail since it is not part of the program consequences.

$$\begin{tikzcd}
 & \langle fac(2,4) \mid true \rangle \arrow[dl, "F1"'] \arrow[dr, "F2"] & \\
\langle \blacksquare \mid 0 = 2 \rangle & & \langle fac(N-1,F) \mid 2 = N \land 4 = N \times F \land N \geq 1 \rangle \arrow[dl, "F1"'] \arrow[dd, "F2"] \\
 & \langle \blacksquare \mid 2 = N \land 4 = N \times F \land N \geq 1 \land N - 1 = 0 \rangle & \\
 & & \begin{array}{c} \langle fac(N1-1,F1) \mid 2 = N \land 4 = N \times F \land N \geq 1 \land \\ N - 1 = N1 \land F = N1 \times F1 \land N1 \geq 1 \rangle \end{array} \arrow[dl, "F1"'] \arrow[dd, "F2"] \\
 & \begin{array}{c} \langle \blacksquare \mid 2 = N \land 4 = N \times F \land N \geq 1 \land N - 1 = N1 \land \\ F = N1 \times F1 \land N1 \geq 1 \land N1 - 1 = 0 \land F1 = 1 \rangle \end{array} & \\
 & & \begin{array}{c} \langle \blacksquare \mid 2 = N \land 4 = N \times F \land N \geq 1 \land N - 1 \land F = N1 \times F1 \land \\ N1 \geq 1 \land N1 - 1 = N2 \land F1 = N2 \times F2 \land N2 \geq 1 \rangle \end{array}
\end{tikzcd}$$

Figure 11.5 Derivation tree for goal `fac(2,4)`.

11.4 Mimicking Top-Down Evaluation Bottom-Up

We have seen that each of the evaluation methods has advantages and disadvantages. Bottom-up evaluation has the advantage of always terminating if the set of program consequences is finite and of recognizing repeated answers. The advantage of top-down evaluation is that it is goal-directed. Evaluation is directed by the initial goal and the goals that are subsequently derived from it. If we are only interested in the answer to a single goal, rather than all of the program consequences, top-down derivation will often be more efficient since it does not bother to determine extraneous information.

To combine the efficiency of both methods, a number of approaches have been suggested for modifying bottom-up evaluation so that it becomes goal-directed. Given a program P and goal G, the key idea is to transform P into *a new program* that, when evaluated bottom-up, will mimic the top-down evaluation of G with the original program.

Example 11.9

We shall first look at how this works for the goal `fac(2, X)` with the factorial program from Example 11.6. Top-down evaluation of the goal gives the derivation tree shown in Figure 11.4.

By examining this derivation tree we can determine the set of calls made to each predicate. We call these the set of ***relevant queries.*** More exactly, for each state $\langle A, B \mid c\rangle$ in the simplified derivation tree, where A is the rewritten user-defined constraint and B is the remainder of the goal, we collect the state $\langle A \mid C\rangle$ which details that user-defined constraint A has been called with constraint store C. These are the queries that top-down evaluation has needed to answer in order to answer the original goal. In this case they are

$$\begin{array}{l}\langle fac(2, X) \mid true\rangle \\ \langle fac(N-1, F) \mid 2 = N \wedge X = N \times F \wedge N \geq 1\rangle \\ \langle fac(N1-1, F1) \mid 2 = N \wedge X = N \times F \wedge N \geq 1 \wedge \\ \qquad N-1 = N1 \wedge F = N1 \times F1 \wedge N1 \geq 1\rangle\end{array}$$

These are the only queries on which the original goal depends. The idea is to transform the factorial program into a new program whose program consequences only contain facts relevant to one of these queries.

To do this we add a filter literal `query_fac` to each rule for `fac` and add rules to construct the relevant queries as facts for the filter predicate. The original factorial program is

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

The result of adding the filter literals is the program QP

```
fac(0, 1) :- query_fac(0, 1).
fac(N, N * F) :- query_fac(N, N * F), N ≥ 1, fac(N-1, F).
query_fac(N-1, F) :- query_fac(N, N * F), N ≥ 1.
query_fac(2, X).
```

To understand the transformation, consider how bottom-up evaluation of QP proceeds. To clarify the relationship with top-down evaluation we will not simplify the facts. After the first iteration, S contains the single fact

```
query_fac(2, X).
```

This corresponds to the original goal in the top-down derivation. The next iteration finds the new fact

```
query_fac(N-1, F) :- 2 = N, X = N * F, N ≥ 1.
```

This corresponds to the second goal in the top-down derivation. The third iteration includes the new fact

```
query_fac(N1-1, F1) :-
     N1 = N-1, N1 * F1 = F,
     2 = N, X = N * F,
     N ≥ 1, N1 ≥ 1.
```

which corresponds to the third goal in the top-down derivation. The fourth iteration produces the fact

```
fac(0, 1) :-
    0 = N1-1, 1 = F1,
    N1 = N-1, N1 * F1 = F,
    2 = N, X = N * F,
    N ≥ 1, N1 ≥ 1.
```

This fact can be simplified to `fac(0,1)`. It may be understood as an answer to the state

$$\langle fac(N1-1, F1) \,|\, 2 = N \wedge X = N \times F \wedge N \geq 1 \wedge \\ N-1 = N1 \wedge F = N1 \times F1 \wedge N1 \geq 1\rangle.$$

Once the answer is computed for this state it can be used to generate an answer to the state

$$\langle fac(N-1, F) \,|\, 2 = N \wedge X = N \times F \wedge N \geq 1\rangle$$

since both a `query_fac` and `fac` fact are available.

Subsequent iterations create the answers to the second and first states. The total set of (simplified) facts produced is shown below, where the BUi label indicates that this fact was first generated in the i^{th} iteration of the main loop.

```
query_fac(2, X),      (BU1)
query_fac(1, F),      (BU2)
query_fac(0, F1),     (BU3)
fac(0, 1),            (BU4)
fac(1, 1),            (BU5)
fac(2, 2).            (BU6)
```

Note that the set of program consequences for the transformed program is finite. Evaluation of the original goal `fac(2, X)` with this set of facts gives the single answer $X = 2$.

In general we can make bottom-up evaluation goal-directed by adding a new predicate *query_p* to the program for each of the original predicates *p*. The new predicate, *query_p*, generates the calls made to *p* during the top-down evaluation of the goal *G*. The query predicates act as filters, preventing bottom-up evaluation from computing facts which are not *relevant* to the queries. The general transformation is defined as follows.

Algorithm 11.2: Query transformation
INPUT: A program P and goal G of form $c_1, \ldots, c_n, q(r_1, \ldots, r_m)$ where each c_i is a primitive constraint and $q(r_1, \ldots, r_m)$ is a user-defined constraint.
OUTPUT: A program QP which, when evaluated bottom-up, mimics the top-down evaluation of G using P.
METHOD: Initially, QP is an empty program.

1. For each predicate p in P, create a new predicate *query_p* with the same arity as p.
2. For each rule in P, add the *modified version* of the rule to QP. If a rule has head $p(s_1, \ldots, s_m)$, the modified version of this rule is obtained by adding the literal $query\_p(s_1, \ldots, s_m)$ to the front of the body.
3. Create a *seed rule*, $query\_q(r_1, \ldots, r_m)$:- $c_1, \ldots, c_n$, and add this to QP.
4. For each rule R in P of the form

$$p(s_1, \ldots, s_m) \text{ :- } L_1, \ldots, L_k$$

and for each literal L_i, $1 \leq i \leq k$ which is a user-defined constraint, add a *query rule* to QP of the form

$$query\_p_i(t_1, \ldots, t_m) \text{ :- } query\_p(s_1, \ldots, s_m), L_1, \ldots, L_{i-1}.$$

where L_i has form $p_i(t_1, \ldots, t_m)$.

□

The transformation rules can be understood as follows. The modified rules restrict the original rules so that they only produce facts that are relevant to the query. That is to say, they have a corresponding query fact. Calculation of the relevant queries is initiated from the seed rule which states that the goal is a relevant query. The remaining relevant queries are deduced by mimicking the top-down computation. There is a relevant query for p_i the i^{th} literal in a rule for p if there is a relevant query for p and answers to all the preceding literals $L_1, \ldots, L_{i-1}$.

Example 11.10
Consider Ackermann's function

$$ack(m,n) = \begin{cases} n+1, & \text{if } m = 0; \\ ack(m-1, 1), & \text{if } m \geq 1, n = 0; \\ ack(m-1, ack(m, n-1)), & \text{otherwise.} \end{cases}$$

The following program is a natural translation of this function with the predicate `ack(M,N,A)` holding whenever $A = ack(M, N)$.

```
ack(M, N, A) :- M = 0, A = N + 1.
ack(M, N, A) :- M ≥ 1, N = 0, ack(M-1, 1, A).
ack(M, N, A) :- M ≥ 1, N ≥ 1, ack(M, N-1, A1), ack(M-1, A1, A).
```

Given this program and the goal `ack(2,1,A)`, application of the query translation algorithm gives rise to the following program

```
ack(M,N,A) :-
     query_ack(M,N,A),
     M = 0, A = N + 1.
ack(M,N,A) :-
     query_ack(M,N,A),
     M ≥ 1, N = 0, ack(M-1,1,A).
ack(M,N,A) :-
     query_ack(M,N,A),
     M ≥ 1, N ≥ 1, ack(M,N-1,A1), ack(M-1,A1,A).

query_ack(2,1,A).
query_ack(M-1,1,A) :-
     query_ack(M,N,A),
     M ≥ 1, N = 0.
query_ack(M,N-1,A1) :-
     query_ack(M,N,A),
     M ≥ 1, N ≥ 1.
query_ack(M-1,A1,A) :-
     query_ack(M,N,A),
     M ≥ 1, N ≥ 1, ack(M,N-1,A1).
```

The advantage of applying the query transformation is that the bottom-up evaluation of the resulting program will compute only those facts that are relevant to the original query. This may improve termination behaviour since, even if the original program does not terminate, the transformed program may. The factorial program gives an example of this behaviour.

In general, since the query transformed program mimics top-down evaluation, it will always terminate when top-down evaluation will terminate (after finding all answers). However, sometimes it will terminate even when top-down evaluation does not. This is because repeated answers are detected. For instance, bottom-up evaluation of the transformed flight database from Example 11.4 with goal `connects(melbourne,T)` will terminate, even though top-down evaluation of this goal will not.

In most cases, application of the query transformation is worthwhile. It improves efficiency by substantially reducing the number of facts that need to be computed and often improves termination behaviour. However, the query transformation is not always beneficial. First, the resulting program is more complex than the original and so may take longer to evaluate if all facts are relevant. Second, very occasionally, bottom-up evaluation of the resulting program may not terminate even though bottom-up evaluation of the original program does.

Example 11.11
Consider the following simple program:

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

Top-down evaluation for the goal `stupid(4)` never halts. Bottom-up evaluation halts after a single iteration since there are no facts. For the goal `stupid(4)`, the query-transformed program is:

```
stupid(X) :- query_stupid(X), stupid(X+1).
query_stupid(4).
query_stupid(X+1) :- query_stupid(X).
```

Bottom-up evaluation of this program will not terminate since an infinite number of query facts will be generated.

11.5 (*) Improving Termination

The bottom-up evaluation algorithm employs the function new(F,S) to determine if information in the fact F already occurs in the old set of facts S or is new. Evaluation terminates when information from all of the "new" facts S_{new} already occurs in S. We now examine two refinements of new which improve the termination behaviour of bottom-up evaluation, although with greater cost in a call to new.

The definition of new given in Figure 11.2 tests whether fact F is equivalent to some fact F_1 in S. If a *canonical form simplifier* (introduced in Section 2.6) is available for the constraint domain of interest, it is simple to implement new using this simplifier. This is because a canonical form simplifier ensures that equivalent constraints are simplified to exactly the same form. To determine if two facts are equivalent, we can, therefore, convert the facts to head normal form and use the canonical simplifier to simplify the body of each in terms of the variables in the head of the fact. If the resulting facts are syntactically identical, then the original facts are equivalent. Otherwise, they are not.

If a canonical form simplifier is not available for our constraint domain we can use an *equivalence tester* (see Section 2.7) to determine equivalence.

However, the definition of new can be improved, so that even if the new facts are not equivalent to the old facts, bottom-up evaluation terminates. This is because the new facts, even though they are not equivalent to old facts, may hold no new information.

Example 11.12
The following simple program over the domain of real numbers computes successively smaller boxes centred on the origin.

```
box(X, Y) :- X ≥ -4, X ≤ 4, Y ≥ -4, Y ≤ 4.
box(X, Y) :- X = X1/2, Y = Y1/2, box(X1, Y1).
```

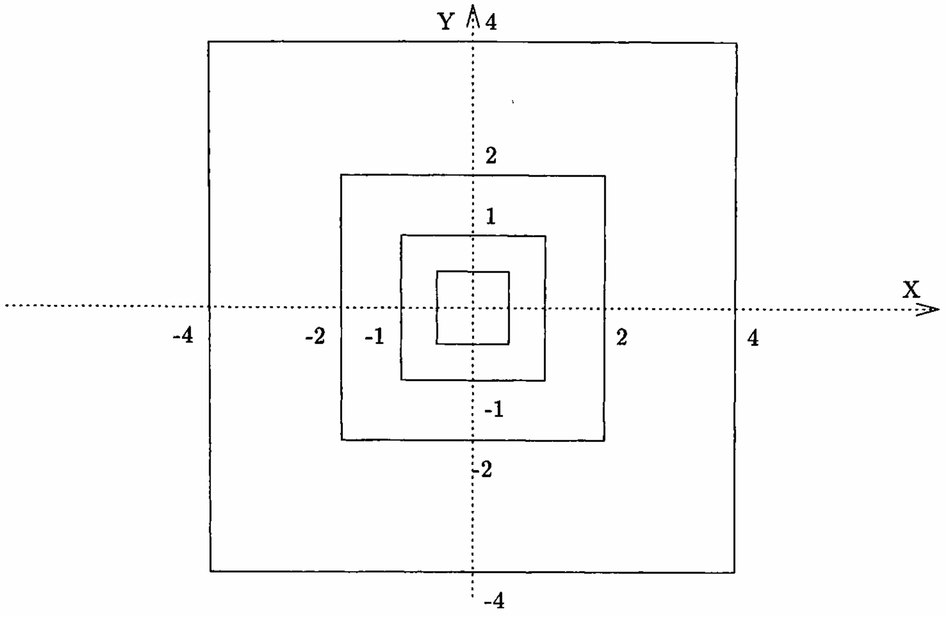

Figure 11.6 Areas representing facts for `box`.

Bottom-up computation produces the facts

```
box(X, Y)  :-  X ≥ -4, X ≤ 4, Y ≥ -4, Y ≤ 4,
box(X, Y)  :-  X ≥ -2, X ≤ 2, Y ≥ -2, Y ≤ 2,
box(X, Y)  :-  X ≥ -1, X ≤ 1, Y ≥ -1, Y ≤ 1,
box(X, Y)  :-  X ≥ -0.5, X ≤ 0.5, Y ≥ -0.5, Y ≤ 0.5,
               ⋮
```

in this order. A diagram of the areas described by the facts is given in Figure 11.6.

The set of program consequences is infinite since it contains arbitrarily small boxes centred at the origin and none of the facts in this set are equivalent. However, all of the "information" in this set is captured in the first fact

```
box(X, Y) :- X ≥ -4, X ≤ 4, Y ≥ -4, Y ≤ 4
```

since each of the other facts is *subsumed* by this fact in the sense that every solution to another fact is also a solution of this fact. Thus bottom-up computation could stop, with no loss of information, after producing the first fact.

To overcome this problem we can use a modified version of bottom-up evaluation. In this modification, a fact is considered new only if it is not redundant with respect to any earlier fact. To make this precise, we need to generalize our earlier notion of constraint implication from Chapter 2 to take into account the variables of interest.

Definition 11.6
Constraint C_1 *implies* constraint C_2 with respect to the set of variables $\{x_1, \dots, x_n\}$ if for each solution $\{x_1 \mapsto d_1, \dots, x_n \mapsto d_n, \dots\}$ of C_1, $\{x_1 \mapsto d_1, \dots, x_n \mapsto d_n\}$ is a partial solution of C_2.

Definition 11.7
Let fact F_1 have head normal form

$$A \text{ :- } c_1, \dots, c_m$$

and fact F_2 have head normal form

$$A' \text{ :- } c'_1, \dots, c'_n.$$

F_1 is *subsumed* by F_2 if A and A' are identical and $c_1 \wedge \cdots \wedge c_m$ implies $c'_1 \wedge \cdots \wedge c'_n$ with respect to the variables in A.

We can replace the call in new to test if two constraints are equivalent by a call to test if one subsumes the other. This gives the definition

```
new($F$, $S$)
    for each $F_1 \in S$ do
        if $F$ is subsumed by $F_1$ then return $false$ endif
    endfor
    return $true$
```

We call the resulting variant of bottom-up evaluation, bottom-up evaluation *with subsumption testing.*

Example 11.13
Bottom-up evaluation with subsumption testing of the program of Example 11.12 and goal box(X,Y) proceeds as follows. The first iteration produces the fact

```
box(X, Y) :- X ≥ -4, X ≤ 4, Y ≥ -4, Y ≤ 4.
```

In the next iteration S is equal to the singleton set containing the above fact and S_{new} is computed to be the set containing this fact as well as the fact

```
box(X, Y) :- X ≥ -2, X ≤ 2, Y ≥ -2, Y ≤ 2
```

which was produced from the second rule. Evaluation of same(S, S_{new}) returns *true* since each fact in S_{new} is subsumed by the single fact in S. Thus the main loop terminates and the algorithm returns the single answer

$$X \geq -4 \wedge X \leq 4 \wedge Y \geq -4 \wedge Y \leq 4.$$

By now the reader might hope that we have conquered all problems with termination. Unfortunately, this is not the case, because seemingly new facts may not be subsumed by a single old fact, but only by a number of old facts considered together.

Example 11.14
The following simple program produces rectangles and quarter-planes over the domain of real numbers.

```
rect(X, Y) :- X ≤ 0, Y ≥ 0.
rect(X, Y) :- X ≤ 0, Y ≤ 0.
rect(X, Y) :- X ≥ 1, X ≤ 2, Y ≥ -1, Y ≤ 1.
rect(X, Y) :- X = X1 + 2, rect(X1, Y).
```

The first iteration produces the facts, S_1

```
rect(X, Y) :- X ≤ 0, Y ≥ 0,
rect(X, Y) :- X ≤ 0, Y ≤ 0,
rect(X, Y) :- X ≥ 1, X ≤ 2, Y ≥ -1, Y ≤ 1.
```

These define two half-planes that together include all of the non-positive values of X, and a rectangle of size 1×2 that is centred around the X axis and starts at X coordinate 1. The next iteration produces in addition to the facts in S_1 the new facts,

```
rect(X, Y) :- X ≤ -2, Y ≥ 0.
rect(X, Y) :- X ≤ -2, Y ≤ 0.
rect(X, Y) :- X ≥ -1, X ≤ 0, Y ≥ -1, Y ≤ 1.
```

The first two facts are subsumed by one of the half planes in S_1. However, although the third fact is not subsumed by any single fact in S_1, it is redundant with respect to the first two facts in S_1. If this redundancy is not detected evaluation will continue, producing the facts

```
 rect(X, Y)  :-  X ≥ -3, X ≤ -2, Y ≥ -1, Y ≤ 1,
 rect(X, Y)  :-  X ≥ -5, X ≤ -4, Y ≥ -1, Y ≤ 1,
                 ⋮
```

and will not terminate.

A diagram of the areas described by the facts is given in Figure 11.7. The shaded areas correspond to the first two facts in S_1, while the rectangles correspond to the third fact in S_1 and the facts produced later which are not subsumed by a single fact.

We can avoid the type of behaviour exhibited in the above example by extending the subsumption check so that it succeeds if a new fact is subsumed by the set of old facts considered together. We leave the definition of the new version of new as an exercise. Unfortunately, such multi-fact subsumption testing is usually very expensive—and thus impractical.

However, in some constraint domains, multi-fact subsumption testing is no harder than testing if a fact is subsumed by a single fact. This is because in such domains, the solutions to a constraint C are covered by the solutions to a number of

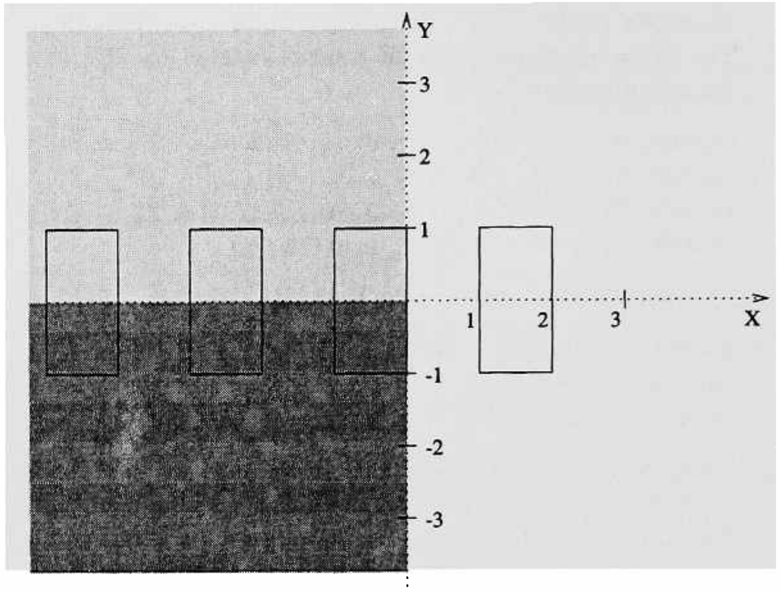

Figure 11.7 Areas representing facts for `rect`.

constraints $C_1, \ldots, C_n$ only if C is subsumed by one of the C_i. Such domains are said to have *independence of negative constraints.* For instance, the tree constraint domain has independence of negative constraints, while, as the above example demonstrated, linear arithmetic constraints do not have independence of negative constraints.

11.6 (*) Relationship with Relational Databases

The relational data model is currently the most popular form of database, in part because its high level nature frees the user from decisions about how queries are to be answered. A constraint database is, arguably, even more high level. In this section we use examples to show that constraint databases are a natural generalization of relational databases.

We assume the reader has some basic understanding of (relational) databases, but shall give the basic definitions to clarify our notation and terminology.

Definition 11.8

A ***relational domain*** is a set of values. The ***Cartesian product*** of (not necessarily disjoint) relational domains $D_1, \ldots, D_k$, written $D_1 \times \cdots \times D_k$, is the set of all k-tuples $(d_1, \ldots, d_k)$ where $d_i \in D_i$ for each $1 \leq i \leq k$. A k-ary ***relation*** is a subset of $D_1 \times D_2 \times \cdots \times D_k$. We may associate with a k-ary relation a k-tuple of ***attributes*** which uniquely names each argument of the relation.

Relational databases store, update and manipulate relations. A *relational database* can be viewed as a (time-varying) collection of relations. Since the database relations are stored and used in computation we assume that they are finite in size.

Example 11.15

Consider a simple database storing information about the teaching operations of a computer science department. Some appropriate relational domains are: courses, teachers, days of the week, times, rooms and integers. We can model our data using two relations. The first is a relation lecture with six attributes which gives information about lectures. It details the course name, lecturer, day, start and finish times and room of each lecture. It is described by the following table in which the attribute names are given at the top of each column in the table.

lecture

Course_Name	Lecturer	Day	Start_Time	Finish_Time	Room
logic	harald	mon	1000	1230	theatre2
theory	harald	tue	1100	1200	theatre2
theory	harald	thu	1100	1200	theatre2
constraints	peter	mon	1215	1315	theatre1
constraints	peter	fri	1515	1615	theatre1

The second is a binary relation enrol which gives the number of students in each course. It is detailed in the following table. Again the table details the attribute names.

enrol

Course_Name	Enrolment
logic	46
theory	27
constraints	142

Queries are made to relations in the database by constructing a relational algebra expression. This defines a new relation in terms of the given relations, and the answer to the query is the tuples of this relation. There are many different possible operations over relations. Here we restrict ourselves to the five most fundamental.

Union: If r_1 and r_2 are two k-ary relations then the *union* of r_1 and r_2, written $r_1 \cup r_2$, is the relation

$$\{(d_1, \ldots, d_k) \mid (d_1, \ldots, d_k) \in r_1 \text{ or } (d_1, \ldots, d_k) \in r_2\}.$$

Projection: If r is a k-ary relation and V is the tuple of attribute identifiers $(\$v_1, \ldots, \$v_n)$ where $v_1, \ldots v_n$ are numbers from the range 1 to k, the *projection* of r on to V, written $\pi_V r$, is the relation

$$\{(d_{v_1}, \ldots, d_{v_n}) \mid (d_1, \ldots, d_k) \in r\}.$$

Selection: If r is a k-ary relation and c is a primitive constraint (from some constraint domain) over the attribute identifiers $\$1, \ldots, \$k$, the *selection* of r using c, written $\sigma_c r$, is the relation

$$\{(d_1, \ldots, d_k) \in r \mid \{\$1 \mapsto d_1, \ldots, \$k \mapsto d_k\} \text{ is a solution of } c\}.$$

Product: If r_1 is a k-ary relation and r_2 is an n-ary relation then the *product* of r_1 and r_2, written $r_1 \times r_2$, is the relation

$$\{(d_1, \ldots, d_k, e_1, \ldots, e_n) \mid (d_1, \ldots, d_k) \in r_1, (e_1, \ldots, e_n) \in r_2\}.$$

Natural join: Let r_1 be a k-ary relation and r_2 an n-ary relation. The *natural join* of r_1 and r_2, written $r_1 \bowtie r_2$, is rather complex to define. We give the definition for the case in which r_1 and r_2 have no attribute names in common and the case when they have a single attribute name in common. The generalization to two or more attribute names in common is straightforward but tedious to formalize.
In the case that r_1 and r_2 have no attribute names in common, $r_1 \bowtie r_2$ is simply the product $r_1 \times r_2$.
In the case that r_1 and r_2 have a single attribute name in common, say attribute i of r_1 and attribute j of r_2 have the same name, then $r_1 \bowtie r_2$ is

$$\{(d_1, \ldots, d_k, e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_n) \mid (d_1, \ldots, d_k) \in r_1, (e_1, \ldots, e_n) \in r_2, d_i = e_j\}.$$

If a k-ary relation r has an associated list of attribute names $(a_1, \ldots a_k)$ we may also use a_i to represent $\$i$ in projections and selections.

Definition 11.9
A relation name is a *relational expression*. Furthermore, if R_1 and R_2 are relational expressions then so are $R_1 \cup R_2$, $\pi_V R_1$, $\sigma_c R_1$, $R_1 \times R_2$ and $R_1 \bowtie R_2$. A *query* on a database D is a relational expression involving only relation names from D.
The evaluation of a query Q on database D is the relation given by the relational expression Q where D gives the meaning of the named relations in Q.

Using relational operations it is simple to ask various questions about the sample database from Example 11.15.

For instance, using the query $\pi_{\$1,\$2}$(lecture) we can ask for the names of the lecturers teaching a course. Alternately, instead of using a numerical identifier to refer to the attributes, we can use the attribute names. This gives rise to the equivalent query $\pi_{Course\_Name,Lecturer}$(lecture). Evaluation of either query gives the relation

Course_Name	Lecturer
logic	harald
theory	harald
constraints	peter

Using the query $\sigma_{\$2 \geq 100}(\text{enrol})$ we can determine which classes have more than 100 students. In this case the corresponding query using attribute names is simply $\sigma_{Enrolment \geq 100}(\text{enrol})$. Evaluation of either query results in the relation

Course_Name	Enrolment
constraints	142

A slightly more complex query is to ask for the classes which are non-standard in size. That is to say, they are large with 100 or more students or small with 30 or less students. The query

$$\sigma_{\$2 \geq 100}(\text{enrol}) \cup \sigma_{\$2 \leq 30}(\text{enrol})$$

will find such classes. Evaluation will return the relation

Course_Name	Enrolment
theory	27
constraints	142

An even more complex query is to ask for the names of lecturers that teach large classes. This requires us to use the natural join to concatenate tuples in the two relations which refer to the same course. The answer can be found by evaluating

$$\pi_{\$4}((\sigma_{\$2 \geq 100}(\text{enrol})) \bowtie (\pi_{\$1,\$2}(\text{lecture})))$$

or, alternately using attribute names, by evaluating

$$\pi_{Lecturer}((\sigma_{Enrolment \geq 100}(\text{enrol})) \bowtie (\pi_{Course\_name,Lecturer}(\text{lecture}))).$$

The result is the relation consisting of a single tuple (*peter*).

It is not too hard to show that any relational database and query can be translated into a constraint database program and goal. We demonstrate this translation by means of several examples.

In a relational database a relation is simply a finite set of tuples. We can naturally represent each relation by a predicate with one argument for each attribute and represent each tuple by a fact in the definition of the predicate.

Consider the database of Example 11.15. It can be represented by the constraint logic program:

```
lecture(logic, harald, mon, 1000, 1230, theatre2).
lecture(theory, harald, tue, 1100, 1200, theatre2).
lecture(theory, harald, thu, 1100, 1200, theatre2).
lecture(constraints, peter, mon, 1215, 1315, theatre1).
lecture(constraints, peter, fri, 1515, 1615, theatre1).

enrol(logic, 46).
enrol(theory, 27).
enrol(constraints, 142).
```

Each of the relational operations—union, selection, projection, product and join—can be naturally modelled using constraint programs. Consider our example queries to the above database.

Projection corresponds to "forgetting" the value of local variables. For instance, the query asking for the names of the lecturers teaching a course, $\pi_{Course\_name,Lecturer}(\textsf{lecture})$ is modelled by the rule

```
teaches(C, L) :- lecture(C, L, D, S, F, R).
```

Answers to the goal `teaches(C, L)` correspond exactly to the tuples in the answer to the original query. If we evaluate the goal `teaches(C, L)` with the above definition of `lecture` we will obtain the three answers

$$C = logic \wedge L = harald,$$
$$C = theory \wedge L = harald,$$
$$C = constraints \wedge L = peter,$$

which correspond to the three tuples resulting from the original query to the database.

Selection is easy to implement using constraints. The query asking for those classes which have more than 100 students, $\sigma_{Enrolment \geq 100}(\textsf{enrol})$, is modelled by the rule

```
large_class(C, E) :- enrol(C, E), E ≥ 100.
```

The answers to the goal `large_class(C, E)` correspond to the answers of the original query.

Union is modelled using multiple rules in the definition of a relation. For instance, the query

$$\sigma_{\$2 \geq 100}(\textsf{enrol}) \cup \sigma_{\$2 \leq 30}(\textsf{enrol})$$

is modelled by

```
non_standard_class(C, E) :- enrol(C, E), E ≥ 100.
non_standard_class(C, E) :- enrol(C, E), E ≤ 30.
```

Natural join is modelled by two user-defined constraints in a body with shared attributes having the same variable as their argument. Consider the query asking for the names of lecturers that teach large classes,

$$\pi_{Lecturer}((\sigma_{Enrolment \geq 100}(\textsf{enrol})) \bowtie (\pi_{Course\_name,Lecturer}(\textsf{lecture}))).$$

Using the above definitions this is modelled by the program

```
teaches(C, L) :- lecture(C, L, D, S, F, R).
large_class(C, E) :- enrol(C, E), E ≥ 100.
teaches_large_class(L) :- large_class(C, E), teaches(C, L).
```

The last rule models the natural join between `large_class` and `teaches` projected on to the attribute *Lecturer*. The goal `teaches_large_class(L)` has a single answer $L = peter$ corresponding to the single tuple in the answer to the query.

Product is modelled similarly to natural join. We simply define a new predicate which has all of the arguments of each of the relations.

It should be apparent from these examples that constraint databases are as powerful as relational databases: any data or query which can be expressed using a relational database can also be expressed using a constraint database. In fact, constraint databases are more powerful than traditional relational databases. There are two reasons. First, constraint databases provide *deduction*. This is provided by recursive rules such as those defining the `connects` relation in Example 11.4. Expressing such a recursive relation is not possible using relational algebra. Second, in applications such as those involving temporal and spatial data, they allow many tuples to be succinctly described by a single constraint.

11.7 Summary

Constraint databases are a distinctive new type of database. The use of constraints provides a uniform framework which generalizes other types of databases such as relational databases, deductive databases and spatial databases.

Here we have viewed constraint databases simply as constraint logic programs. However, one important difference is that, in constraint logic programming, we are usually interested in obtaining only one answer to our goal, while in constraint databases we usually want all of the answers. For this reason we have introduced a new kind of evaluation mechanism for constraint logic programs, called bottom-up evaluation, which iteratively computes a set of facts called the program consequences.

Bottom-up evaluation has advantages and disadvantages when compared with the standard top-down evaluation of CLP programs. The main advantage of top-down evaluation is that it is goal-directed. To combine the advantages of bottom-up with that of top-down, we have detailed a transformation that, when applied to a program and goal, gives a new program whose bottom-up evaluation is goal-directed and mimics the top-down evaluation of the original program.

One advantage bottom-up evaluation has over top-down evaluation is that it can include sophisticated checks for termination. We have discussed various possibilities including equivalence, subsumption testing and multi-fact subsumption testing.

Our presentation has ignored many interesting issues in constraint databases. The bottom-up evaluation strategy we have defined is quite simplistic, repeatedly evaluating the entire program in each iteration. More efficient evaluation methods which do not repeat computation unnecessarily are usually used for evaluating recursive constraint database queries. We have also ignored many important database operations, such as negation and aggregation, on constraint databases and concentrated only on constraint logic programs as a database language.

11.8 Exercises

11.1. Show the bottom-up evaluation of the program in Example 11.3.

11.2. Give the query transformation of the `mortgage` program of Section 5.3 for the goal `mortgage(2000, 2, 10, R, 0)`. Show the bottom-up evaluation of the resulting program.

11.3. Using the genealogical database and rules defining `parent` from Example 11.1 and the additional rules

```
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X ≠ Y.
cousin(X, Y) :- parent(Z, X), sibling(Z, T), parent(T, Y).
```

show the bottom-up evaluation of the query `cousin(peter, X)`. Perform the query transformation for this query and re-evaluate. Compare the number of facts in the program consequences for each evaluation.

11.4. Perform the query transformation for the program below with the goal `p(3)` and then with `X ≤ 4, p(X)`.

```
p(0).
p(X) :- X ≥ 1, p(X-1).
```

Determine which form of the routine *new* is required to ensure the termination of bottom-up evaluation of each of the resulting programs.

11.5. (*) Give a relational expression that finds pairs of courses whose lectures overlap in time. Give a corresponding constraint program.

11.9 Practical Exercises

P11.1. Write a query to determine the times when two lecturers $L1$ and $L2$ are teaching at the same time. Test it with the data for the lecture relation.

P11.2. Write a CLP program which will terminate using top-down evaluation and which defines the `connects(X,Y)` predicate for Example 11.4. [Hint: use an extra argument which keeps the list of cities already visited and checks that each new city has not been previously visited.]

P11.3. (*) Write a program to perform bottom-up evaluation of CLP(*Tree*) programs as described in Algorithm 11.1. Make the *new* routine as complete as possible.

P11.4. (*) Write a program to perform the query transformation to a program P and goal G. Use the `lrule` representation of programs described in Section 9.7. Check that the resulting program QP gives the same answers as the original program P for goal G.

11.10 Notes

The bottom-up evaluation viewpoint of constraint logic programs is implicit in the original paper on constraint logic programming [69]. Kanellakis, Kuper and Revesz [77] were the first to formally define a *constraint database* in terms of "generalized tuples" (corresponding to facts in this presentation) and proposed a framework for constraint databases using this approach.

Although phrased somewhat differently, equivalence between bottom-up and top-down evaluation is shown in Jaffar and Lassez [69]. The query transformation is a simplified version of the Magic Templates transformation of Ramakhrishnan [108], which is a generalization of the Magic Sets transformation [7].

Bottom-up evaluation is more efficient if performed in a differential or semi-naive method [6] in which in each iteration a rule is only used to produce more facts if a new fact for a literal in its body was created in the previous iteration.

Currently, constraint databases implementations are rare. This is because it is a new field and, since it requires much of the machinery of relational and deductive databases, implementation of constraint databases is time-consuming. The best example is perhaps the C^3 system [18]. This constraint database is built in C++ using the object oriented database ObjectStore to manage persistent storage of constraint relations and the linear programming package CPlex as a constraint solver. The database provides an SQL-like language for querying the constraint relations. The query language does not allow recursive queries but instead provides a powerful notion of iterators that can be used to define very complex forms of queries without recursion. This has the advantage that termination is guaranteed without special checks.