1 Constraints

Constraint programming is built upon constraints and constraint solving. Therefore, some knowledge of different constraint domains and algorithms for constraint manipulation is needed to fully understand the constraint programming paradigm.

In the first part of this book we introduce the three most important types of constraints in constraint programming: arithmetic constraints, tree constraints and finite domain constraints. We also investigate three fundamental operations involving constraints. The most important operation is to determine if a constraint is satisfiable. The second operation, simplification, rewrites a constraint in a form which makes its information more apparent. The third operation, optimization, finds a solution which is "best" according to some criterion.

In this chapter we introduce constraints and study the most basic question we can ask about a system of constraints—is it "satisfiable," that is to say, does the system have a solution? Constraints are a mathematical formalization of relationships which can hold between objects. For example "next to" or "father of" are constraints which hold between objects in the real world. We will be chiefly concerned with more idealized constraints between mathematical objects, such as numbers. Real world constraints and objects can be modelled by these idealized mathematical constraints. Building such models is the job of the constraint programmer.

1.1 Constraints and Valuations

The statement $X = Y + 2$ is an example of a *constraint*. X and Y are *variables*, that is they are ***place-holders*** for values, presumably in this case numbers. It stipulates a relation which must hold between any values with which we choose to replace X and Y, namely, the first must be two greater than the second. The statement $X = Y + 2$ is not simply true or false. Its status depends on the values we substitute for X and Y. However, when we have a statement involving variables, one question we can meaningfully ask is whether *any* substitution will yield truth. In the case of $X = Y + 2$ the answer is clearly *yes* (take $X = 3$ and $Y = 1$, for example), but in other cases the answer will be *no*, witness $X = X + 2$. Occasionally the answer may even be *don't know*!

Throughout this book we shall use strings which start with an upper case letter or an underscore "_" to name variables. For example: X, Y, CI, T_A, *List*, *_X*, *_List* are all variables.

Constraints occur in all sorts of everyday reasoning. Consider the following constraint which describes a simple loan. Let P be the principal of the loan (the amount to be borrowed), let I be the interest rate of the loan (so $I \times P$ is the amount of interest to be paid), and let B be the final balance of the loan (the amount owed at the end). Then the constraint

$$B = P + I \times P$$

describes the relationship forced to hold between these variables in a simple loan. For example, if the principal P is \$1000 and the interest rate I is 12% (that is 12/100), then the balance must be \$1120. The constraint imposes a value on the balance given the principal and interest rate, but it also works in other ways. Suppose the balance B is \$2100 and the principal P is \$2000, then the interest rate I must be 5% or 5/100, because it is the only number that satisfies the relationship

$$2100 = 2000 + I \times 2000.$$

The legitimate forms of constraint and their meaning is specified by a ***constraint domain***. Constraints are written in a language made up of constants like 0 and 1, functions like $+$ and $\times$, and constraint relations like equality ($=$) and less-than-or-equal-to ($\leq$).

The constraint domain specifies the "syntax" of the constraints. That is, it specifies the rules for creating constraints in the domain. It details the allowed constants, functions and constraint relations as well as how many arguments each function and constraint relation is required to have and how the arguments are placed. For instance, both $+$ and $\leq$ require two arguments and are written between their arguments.

The constraint domain also determines the values that variables can take. For example, the constraint $X \times X \leq 1/2$ means quite different things depending on whether X can take integer values, real values, or complex values.

Finally, the constraint domain determines the meaning of all of these symbols. It determines what will be the result of applying a function to its arguments and whether a constraint relation holds for given arguments. For example, for arithmetic the application of the $+$ function in the expression $3 + 4$ evaluates to 7, and the constraint $1 \leq 2$ holds while it is not the case that $2 \leq 1$ holds.

Given a constraint domain $\mathcal{D}$ the simplest form of constraint we can define is a ***primitive constraint***. A primitive constraint consists of a constraint relation symbol from $\mathcal{D}$ together with the appropriate number of arguments. These are constructed from the constants and functions of $\mathcal{D}$ and variables. Most constraint relations are binary, that is they require two arguments.

For example, in the constraint domain of real numbers, which we call *Real*, variables take real number values. We shall use *Real* as the default constraint domain throughout most of the text. The set of function symbols for this constraint domain is $+$, $\times$, $-$ and $/$, while the set of constants is all the floating point numbers. The constraint relation symbols are $=, <, \leq, >, \geq$, all of which take two arguments. $X > 3$ and $X + Y \times Y - 3 \times Z = 5$ are examples of primitive constraints.

More complicated constraints can be built from primitive constraints by using the conjunctive connective $\wedge$ which stands for "and". Consider a second year of the loan. The balance after two years, $B2$, is related to the balance after the first year, B, and the interest rate I, as follows: $B2 = B + I \times B$. Hence the entire two year loan is described by:

$$B = P + I \times P \;\wedge\; B2 = B + I \times B.$$

We can use the constraint to calculate values of interest. For example, if the initial principal is \$1000, and the interest rate is 12%, then the balance after two years, $B2$, is \$1254.40. In general, constraints are just sequences of primitive constraints, joined by conjunction.

Definition 1.1
A *constraint* is of the form $c_1 \wedge \cdots \wedge c_n$ where $n \geq 0$ and $c_1, \ldots, c_n$ are primitive constraints. The symbol $\wedge$ denotes *and*, so a constraint $c_1 \wedge \cdots \wedge c_n$ holds whenever all of the primitive constraints $c_1, \ldots, c_n$ hold.
There are two distinct constraints *true* and *false*. The constraint *true* always holds, while *false* never holds. The empty conjunction of constraints (when $n = 0$) is written as *true*.
The *conjunction* of two constraints C_1 and C_2, written $C_1 \wedge C_2$ is defined to be

$$c_1 \wedge \cdots \wedge c_n \wedge c'_1 \wedge \cdots \wedge c'_m$$

where C_1 is $c_1 \wedge \cdots \wedge c_n$ and C_2 is $c'_1 \wedge \cdots \wedge c'_m$.

Given a constraint, we can determine values of variables for which the constraint holds. By replacing variables by their values, the constraint can be simplified to a variable-free constraint which may then be seen to be either true or false. For example, consider the constraint

$$B = P + I \times P \wedge \; B2 = B + I \times B.$$

The assignment of values to variables

$$\{P \mapsto 1000, I \mapsto 20/100, B \mapsto 1200, B2 \mapsto 1440\}$$

makes the constraint take the form

$$1200 = 1000 + 20/100 \times 1000 \wedge 1440 = 1200 + 20/100 \times 1200$$

which after some simplification can be seen to be true. The assignment

$$\{P \mapsto 0, I \mapsto 1, B \mapsto 1, B2 \mapsto 1\}$$

leaves the constraint in the form

$$1 = 0 + 1 \times 0 \land 1 = 1 + 1 \times 1$$

which is false.

Definition 1.2
A *valuation* θ for a set V of variables is an assignment of values from the constraint domain to the variables in V. Suppose $V = \{X_1, \ldots, X_n\}$ then θ may be written $\{X_1 \mapsto d_1, \ldots, X_n \mapsto d_n\}$ indicating that each X_i is assigned the value d_i.
An expression e over variables V is given a value $\theta(e)$ under the valuation θ over variables V. $\theta(e)$ is obtained by replacing each variable by its corresponding value and calculating the value of the resulting variable free expression. Let $vars(e)$ denote the set of variables occurring in an expression e. Similarly, let $vars(C)$ denote the set of variables occurring in a constraint C. If θ is a valuation for $V \supseteq vars(C)$, then it is a *solution* of C if $\theta(C)$ holds in the constraint domain.
A constraint C is *satisfiable* if it has a solution. Otherwise it is *unsatisfiable*.

For example, the value of the expression $P + I \times P$ under the valuation $\{P \mapsto 1000, I \mapsto 20/100\}$ is 1200. The valuation $\{X \mapsto 7\}$ is a solution of the constraint $X \geq 4$. Similarly $\{X \mapsto 5, Y \mapsto 7\}$ is a solution of $X \geq 4$, but $\{X \mapsto 0\}$ is not a solution. This constraint, since it has solutions, is satisfiable. In contrast, the constraint

$$X \leq 3 \land X = Y \land Y \geq 4$$

has no solutions—it is unsatisfiable.

We shall consider constraints as pieces of syntax, that is strings of symbols. Thus, for example, $X = Y$ and $Y = X$ are different constraints. We consider a constraint to be a sequence of primitive constraints, so bracketing is not required. For example, $(X = 0 \land Y = 1) \land Z = 2$ and $X = 0 \land (Y = 1 \land Z = 2)$ represent the same constraint. The order of constraints, however, does matter. For example, $X = 0 \land Y = 1$ is not the same constraint as $Y = 1 \land X = 0$. But even though these two constraints are different, clearly they represent the same information. Similarly $X = Y$ and $Y = X$ represent the same information. The following definition captures what it means for two constraints to contain the same information.

Definition 1.3
Two constraints C_1 and C_2 are *equivalent*, written $C_1 \leftrightarrow C_2$, if they have the same set of solutions.

The reason we consider constraints to be sequences of primitive constraints is because the order of the primitive constraints will prove to be important for some of the constraint manipulation algorithms we shall define later. Many algorithms, however, produce the same result, independent of the order and the number of occurrences of each primitive constraint. That is, in these algorithms constraints are treated as sets of primitive constraints. We shall find it useful to have a function which takes a constraint and returns the set of primitive constraints in the constraint:

Definition 1.4
The function *primitives* takes a constraint $c_1 \land \cdots \land c_n$ and returns the set of primitive constraints $\{c_1, \ldots, c_n\}$. That is to say,

$$primitives(c_1 \land \cdots \land c_n) = \{c_1, \ldots, c_n\}.$$

Clearly, if $primitives(C_1) = primitives(C_2)$ then $C_1 \leftrightarrow C_2$.

For example, $primitives(X = 0 \land Y = 1 \land X = 0)$ and $primitives(Y = 1 \land Y = 1 \land X = 0)$ are both $\{X = 0, Y = 1\}$. Not all equivalent constraints contain the same set of primitive constraints. For instance, $X \geq 0 \land Y = X + 2$ is equivalent to $X = Y - 2 \land Y \geq 2$.

Satisfiability and equivalence are some of the most basic questions we can ask about constraints and we shall return to these in Section 1.3 and Chapter 2 respectively. We first investigate why constraints are useful.

1.2 Modelling with Constraints

Constraints are used to model the behaviour of systems of objects in the real world by capturing an idealised view of the interaction among the objects. It is the job of the constraint programmer to simplify a real world situation into a system of constraints about idealised objects, and use this system of constraints to better understand the behaviour of the real world. The trick is to abstract at the right level—too much abstraction and the constraints lose the essence of the real world problem, too little abstraction and the constraints are too difficult to understand. The laws of physics are a well-known example of this type of modelling.

As an example, we shall now look at how electric circuits can be modelled by constraints. Ohm's Law,

$$V = I \times R$$

describes the relationship among voltage V, current I and resistance R in a resistor. Kirchhoff's Voltage Law (KVL) states that the sum of the voltages around any loop in a circuit must equate to zero. Dually Kirchhoff's Current Law (KCL) states that the sum of currents entering a junction in a circuit must equate to zero.

Consider the electric circuit shown in Figure 1.1. Using the above laws we can model the behaviour of the circuit. The following list gives the primitive constraints derived from the circuit together with their justification:

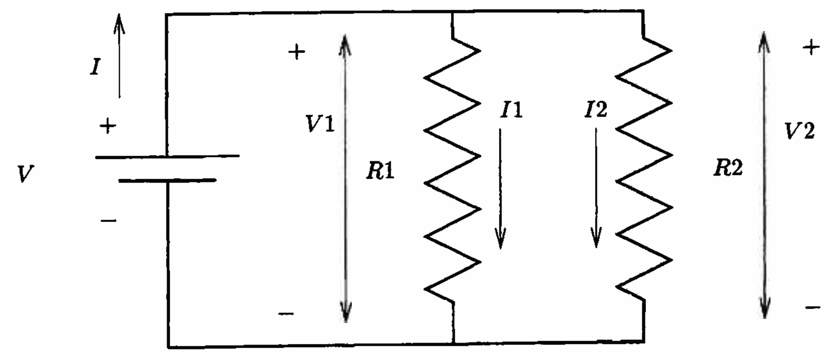

Figure 1.1 A simple electric circuit.

$V_1 = I_1 \times R_1$	Ohm's Law for resistor 1,
$V_2 = I_2 \times R_2$	Ohm's Law for resistor 2,
$V - V_1 = 0$	KVL around the loop (V, R_1),
$V - V_2 = 0$	KVL around the loop (V, R_2),
$V_1 - V_2 = 0$	KVL around the loop (R_1, R_2),
$I - I_1 - I_2 = 0$	KCL at the top junction,
$-I + I_1 + I_2 = 0$	KCL at the bottom junction.

The conjunction of these primitive constraints models the behaviour of the complete circuit. Note that the same constraint information is repeated in some parts of the description, for example, $I - I_1 - I_2 = 0$ and $-I + I_1 + I_2 = 0$ are equivalent.

Given the constraint describing the circuit, we can investigate the circuit's behaviour under various conditions. Suppose the battery is 10V and the resistors $R_1 = 10\Omega$ and $R_2 = 5\Omega$. Then the constraint

$$\begin{array}{l} V = 10 \land R_1 = 10 \land R_2 = 5 \land \\ V_1 = I_1 \times R_1 \land V_2 = I_2 \times R_2 \land \\ V - V_1 = 0 \land V - V_2 = 0 \land V_1 - V_2 = 0 \land \\ I - I_1 - I_2 = 0 \land -I + I_1 + I_2 = 0 \end{array}$$

describes the circuit. This constraint has the single solution

$$\{V \mapsto 10, R_1 \mapsto 10, R_2 \mapsto 5, V_1 \mapsto 10, V_2 \mapsto 10, I_1 \mapsto 1, I_2 \mapsto 2, I \mapsto 3\}.$$

Thus, if we are interested in the current, I, drawn from the battery in this situation, we find it is 3A.

Similarly, we can determine the behaviour when I_1 is known to be 5A, V is known to be 10V and R_2 is known to be 10Ω. In this case there is again a single solution

$$\{V \mapsto 10, R_1 \mapsto 2, R_2 \mapsto 10, V_1 \mapsto 10, V_2 \mapsto 10, I_1 \mapsto 5, I_2 \mapsto 1, I \mapsto 6\}.$$

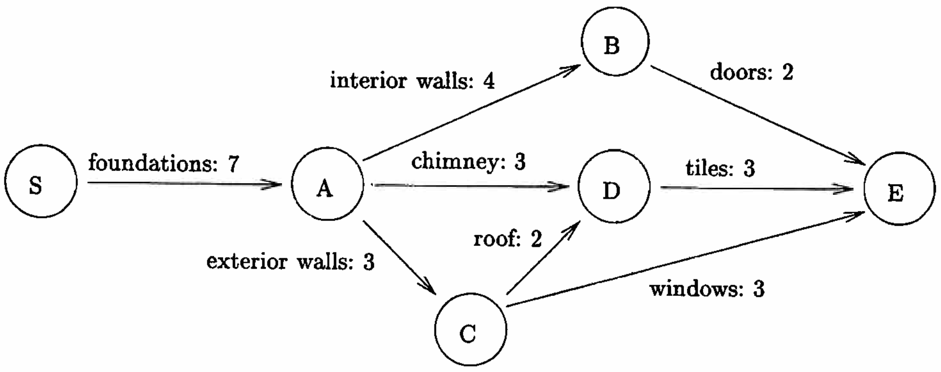

Figure 1.2 Building a house.

If we ask for the behaviour when V is 10V, R_1 is 5Ω, I is 1A and knowing only that the resistance of R_2 is some positive value, that is, $R_2 \geq 0$, we find there are no solutions, since the constraint

$$\begin{aligned}
&V = 10 \wedge R_1 = 5 \wedge I = 1 \wedge R_2 \geq 0 \wedge \\
&V_1 = I_1 \times R_1 \wedge V_2 = I_2 \times R_2 \wedge \\
&V - V_1 = 0 \wedge V - V_2 = 0 \wedge V_1 - V_2 = 0 \wedge \\
&I - I_1 - I_2 = 0 \wedge \ -I + I_1 + I_2 = 0
\end{aligned}$$

describes an impossible situation, and so is unsatisfiable.

Information can be extracted from constraints even when there is more than one solution. Suppose we have that $R_1 = 10\Omega$ and $R_2 = 5\Omega$. In every solution to

$$\begin{aligned}
&R_1 = 10 \wedge R_2 = 5 \wedge \\
&V_1 = I_1 \times R_1 \wedge V_2 = I_2 \times R_2 \wedge \\
&V - V_1 = 0 \wedge V - V_2 = 0 \wedge V_1 - V_2 = 0 \wedge \\
&I - I_1 - I_2 = 0 \wedge \ -I + I_1 + I_2 = 0
\end{aligned}$$

we find that the relationship $3 \times V = 10 \times I$ holds. We will later discuss how to obtain such information.

Constraints are not only useful for modelling physical systems. They may also be used to naturally model other types of systems, for example those occurring in project management. Consider building a house. This project can be broken down into smaller tasks such as: laying foundations, building interior and exterior walls, building the chimney and roof, fitting doors and windows and tiling the roof. Natural restrictions apply to the ordering of these tasks. For example, the roof must be built before it can be tiled. If each task is assigned a duration we can represent the relationships by the graph shown in Figure 1.2 where each arc represents a task, labelled by its name and duration in days, and the nodes represent stages in the project.

To reach a particular stage in the project we need to have completed all the tasks that point to that stage. This means for each such task we must have reached the stage required to begin the task at least a number of days before equal to the duration of the task. If we associate a variable T_s with the time for the earliest arrival at stage s in the project, we can write primitive constraints corresponding to each arc in Figure 1.2 that reflect this information as follows:

start	$T_S \geq 0$,
foundation	$T_A \geq T_S + 7$,
interior walls	$T_B \geq T_A + 4$,
exterior walls	$T_C \geq T_A + 3$,
chimney	$T_D \geq T_A + 3$,
roof	$T_D \geq T_C + 2$,
doors	$T_E \geq T_B + 2$,
tiles	$T_E \geq T_D + 3$,
windows	$T_E \geq T_C + 3$.

A solution of the constraint obtained by conjoining the above primitive constraints represents a possible time-line for the project, that is, a time when each of the stages of the project may be reached. For example

$$\{T_S \mapsto 0, T_A \mapsto 7, T_B \mapsto 11, T_C \mapsto 10, T_D \mapsto 12, T_E \mapsto 15\}$$

is a solution.

Now imagine we are the house builder and that we have promised to have the house ready in two weeks. Unfortunately for us, the constraint above conjoined with the primitive constraint $T_E \leq 14$ yields an unsatisfiable constraint. Thus it is impossible to finish within two weeks, and we had better inform the people paying for the house.

1.3 Constraint Satisfaction

As we have seen, given a constraint, the natural question to ask is: "What are the solutions of the constraint?" Since there may be many solutions to a constraint, this question is usually modified to either the ***solution problem*** "Give me a solution to the constraint if one exists" or the ***satisfaction problem*** "Does the constraint have a solution?"

These two questions are closely related. In a sense, the satisfaction problem is more basic since an answer to the solution problem also answers the satisfaction problem. We now examine methods for answering the satisfaction problem. An algorithm for determining the satisfaction of a constraint is called a ***constraint solver***. We shall see that algorithms for determining constraint satisfaction often construct a solution as a by-product and so can also be used to give a solution.

The obvious way to answer the satisfaction (and solution) question is to enumerate the valuations for the constraint and test whether any is a solution. One rather large drawback of this approach is that there may be an infinite number of valuations to test. Consider the constraint $X > Y$, where X and Y are natural numbers. Checking each possible valuation in the following order never actually finds a solution to the constraint although one exists.

$$\begin{array}{ll} \{X \mapsto 1, Y \mapsto 1\} & \mathit{false} \\ \{X \mapsto 1, Y \mapsto 2\} & \mathit{false} \\ \{X \mapsto 1, Y \mapsto 3\} & \mathit{false} \\ \vdots & \end{array}$$

For the natural numbers, it is possible to enumerate all possible solutions in an order that is guaranteed to find a solution in finite time if one exists. For example,

$$\begin{array}{ll} \{X \mapsto 1, Y \mapsto 1\} & \mathit{false} \\ \{X \mapsto 1, Y \mapsto 2\} & \mathit{false} \\ \{X \mapsto 2, Y \mapsto 1\} & \mathit{true} \\ \{X \mapsto 1, Y \mapsto 3\} & \mathit{false} \\ \{X \mapsto 2, Y \mapsto 2\} & \mathit{true} \\ \{X \mapsto 3, Y \mapsto 1\} & \mathit{true} \\ \{X \mapsto 1, Y \mapsto 4\} & \mathit{false} \\ \vdots & \end{array}$$

this sequence finds a solution at the third attempt and could stop at that stage. However, there is no way to determine in finite time that a given constraint is unsatisfiable. For real number constraints there is no way to enumerate all the possible solutions. Hence no method based on considering all valuations is possible. For constraint domains which have only a finite number of values, we will see in Chapter 3 that enumeration approaches are important.

The problem with enumeration methods is that they only use the constraints in a passive manner, to test the result of applying a valuation, rather than using them to help construct a valuation which is a solution. Of course the way in which the constraints are used depends very much on the type of constraint, therefore, details of how the constraint solver works are usually quite specific to the constraint domain.

One constraint domain that has been intensively studied, is the so called "linear" arithmetic constraints. They are defined as follows. A *linear expression* is of the form $a_1x_1 + a_2x_2 + \cdots + a_nx_n$ where each $a_i, 1 \leq i \leq n$ is a number and each $x_i, 1 \leq i \leq n$ is a variable. A *linear equation* is of the form $e_1 = e_2$ where e_1 and e_2 are linear expressions. Similarly a *linear inequality* is of the form $e_1 \leq e_2$, $e_1 < e_2$, $e_1 \geq e_2$ or $e_1 > e_2$ where e_1 and e_2 are linear expressions. A *linear constraint* is a conjunction of linear equations and inequalities.

One general technique used by many constraint solvers is to repeatedly rewrite a constraint into an equivalent constraint until a constraint in "solved" form is obtained. A ***solved form*** constraint has the property that it is clear whether it is satisfiable or not. For historical reasons such solvers are called ***normalising*** solvers.

We now give an example of a normalising constraint solver for testing satisfiability of conjunctions of linear real equations. The solver uses ***Gauss-Jordan elimination***, which you probably met in high school. The principal step in Gauss-Jordan elimination is to take an equation c, rewrite it into the form $x = e$ where x is a variable and e is a linear expression not involving x, and replace every other occurrence of x by e. Consider the following conjunction of equations:

$$\begin{aligned} 1+X &= 2Y+Z \land \\ Z-X &= 3 \land \\ X+Y &= 5+Z. \end{aligned}$$

Selecting the first equation, we can rewrite it into the form $X = 2Y + Z - 1$. Replacing every other occurrence of X we obtain

$$\begin{aligned} X &= 2Y+Z-1 \land \\ Z-2Y-Z+1 &= 3 \land \\ 2Y+Z-1+Y &= 5+Z. \end{aligned}$$

Examining the second equation, it simplifies to $-2Y = 2$, so we can rewrite it to $Y = -1$. Replacing Y everywhere else by -1, we obtain

$$\begin{aligned} X &= Z-3 \land \\ Y &= -1 \land \\ -2+Z-1-1 &= 5+Z. \end{aligned}$$

The final equation simplifies to $-4 = 5$, which is an unsatisfiable equation. As this final system of equations is equivalent to the initial system we have determined that the original constraint is unsatisfiable.

If we ignore the last equation the resulting system is $X = Z - 3 \land Y = -1$. This is clearly satisfiable since we can choose an arbitrary value for Z (say 4) and then calculate values for Y and X using the equations. The reader may verify that $\{X \mapsto 1, Y \mapsto -1, Z \mapsto 4\}$ is a solution of $1 + X = 2Y + Z \land Z - X = 3$.

In general, a conjunction of equations is in ***solved form*** if it has the form

$$x_1 = e_1 \land x_2 = e_2 \land \cdots \land x_n = e_n,$$

where each of the variables $x_1, \ldots, x_n$ (the *non-parameters*) is distinct and none of them appear in any of the expressions $e_1, \ldots, e_n$. For a large class of constraint domains, including linear real equations, if the variables occurring in $e_1, \ldots, e_n$ (the *parameters*) are each assigned an arbitrary value by valuation θ, then the constraint can be satisfied by assigning to each x_i the value of the expression e_i under θ.

```
$r$ is a real number;
$x$ is a variable;
$e$ is a linear arithmetic expression;
$c$ is an equation;
$C, C_0$ and $S$ are conjunctions of equations.

gauss_jordan_solve($C$)
   if gauss_jordan($C$) $\equiv$ $false$ then return $false$
   else return $true$
   endif

gauss_jordan($C$)
   $S := true$
   while $C$ is not the empty conjunction $true$
      let $C$ be of the form $c \wedge C_0$
      $C := C_0$
      if $c$ can be written in a form without variables then
         if $c$ can be written as $0 = r$ where $r \neq 0$ then
            return $false$
         endif
      else
         write $c$ in the form $x = e$ where $e$ does not involve $x$
         substitute $e$ for $x$ throughout $C$ and $S$
         $S := S \wedge x = e$
      endif
   endwhile
   return $S$
```

Figure 1.3 Gauss-Jordan elimination.

The Gauss-Jordan elimination algorithm has two sets of equations: C, the unsolved equations and S, the solved form equations. At each step the constraint $C \wedge S$ is equivalent to the initial constraint. The algorithm works by repeatedly selecting an equation c from C. If c has no variables it is tested for satisfiability. Otherwise it can be rewritten into the form $x = e$ and used to eliminate x from the remaining equations in C and S. The equation $x = e$ is then added to S.

Note that in this and subsequent algorithms we consider an empty conjunction to be the same as *true*. We use the symbol $\equiv$ to denote syntactic equivalence, that is, $x \equiv y$ holds if x and y are syntactically identical.

Algorithm 1.1: Gauss-Jordan elimination
INPUT: A conjunction of linear real arithmetic equations C.
OUTPUT: Returns *true* if C is satisfiable, otherwise *false*.
METHOD: The algorithm is shown in Figure 1.3. □

$$\begin{tikzcd}
 & & & \textit{order} \arrow[dll] \arrow[d] \arrow[drr] & & & \\
 & \textit{part} \arrow[dl] \arrow[dr] & & \textit{quantity} \arrow[d] & & \textit{date} \arrow[dl] \arrow[d] \arrow[dr] & \\
77665 & & \textit{widget} \arrow[d] \arrow[dr] & 17 & 3 & \textit{feb} & 1994 \\
 & & \textit{red} & \textit{moose} & & &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \textit{cons} \arrow[dl] \arrow[dr] & & & & \\
\textit{red} & & \textit{cons} \arrow[dl] \arrow[dr] & & & \\
 & \textit{blue} & & \textit{cons} \arrow[dl] \arrow[dr] & & \\
 & & \textit{yellow} & & \textit{cons} \arrow[dl] \arrow[dr] & \\
 & & & \textit{grey} & & \textit{nil}
\end{tikzcd}$$

Figure 1.4 Two trees.

1.4 Tree Constraints

Until now, the only constraints we have considered have been arithmetic constraints over the real numbers. However, arithmetic constraints are only one sort of constraint. In this section we introduce another important constraint domain, the domain of *tree constraints* which we shall call *Tree*. Tree constraints allow us to model data structures commonly used in programming, such as records, lists and trees.

Definition 1.5
A *tree constructor* is a string of characters beginning with a lower case letter. A *constant* is a tree constructor or a number.
A *tree* is defined recursively as follows. A constant is a tree, and a tree constructor together with an ordered list of $n \geq 1$ trees, which are called its *children*, is a tree.

Trees are usually drawn with a tree constructor above its children which are displayed in left-to-right order. Arcs are drawn from the tree constructor to each of its children. Figure 1.4 shows some examples of trees. Note that, in the context of trees, numbers such as 17 and 3 are simply constants.

Trees are important because they allow information to be packaged in a structured way. The left tree in Figure 1.4 can be thought of as representing an order for a company. In this case the order is for part number #77665, called a red moose widget, the quantity required is 17 and the date of the order is the 3rd of February 1994. The right tree is a representation of the list $[red, blue, yellow, grey]$. The *cons* tree constructor is the means of connecting the elements of the list together. The constant *nil* signals the end of the list, or rather, stands for the empty list. Trees allow us to express many data structures that are common in programming, such as records, lists and binary trees.

Consider the following C data structure definition for the binary tree data structure:

```
struct tnode {
    struct tnode *left;
    data item;
    struct tnode *right; };
```

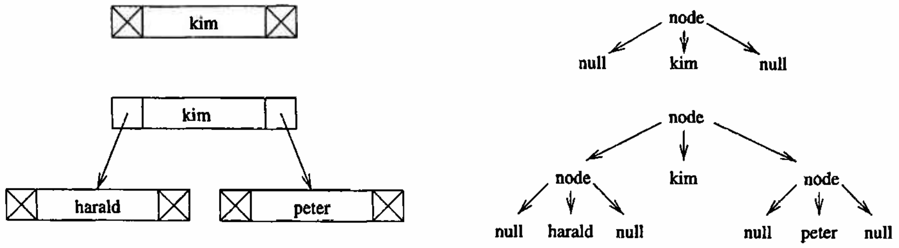

Figure 1.5 Binary trees (a) in C and (b) using trees.

Diagrams of a C binary tree data structure containing the item "kim," and a C binary tree data structure containing the items "harald", "kim" and "peter" are shown in Figure 1.5 (a). A crossed-out pointer box indicates a null pointer. Corresponding trees are shown in Figure 1.5 (b). Note that *null*, *kim*, *peter* and *harald* are all constant trees (with no children), and each tree with the constructor, *node*, has exactly 3 children.

The above definition of trees allows for infinite trees. For example, we can build an infinite tree using a tree constructor f with a single child which has the same form. This gives an infinite chain of f's. However, we will restrict attention to finite trees.

Definition 1.6
A constant a has *height* 1. A tree built from tree constructor f and child trees $t_1, \ldots t_n$ has height equal to one more than the maximum of the heights of the subtrees $t_1, \ldots, t_n$.
A tree is *finite* if it has finite height. Otherwise it is *infinite*.

The bottom tree in Figure 1.5 (b) is of height 2, while the left tree in Figure 1.4 is of height 4. We can write finite trees textually by appropriate bracketing. We first write the tree constructor, then within parentheses a comma separated list of its child trees. For example, the bottom tree of Figure 1.5 (b) can be written

$$node(node(null, harald, null), kim, node(null, peter, null)),$$

the left tree of Figure 1.4 can be written

$$order(part(77665, widget(red, moose)), quantity(17), date(3, feb, 1994)),$$

and the right tree can be written

$$cons(red, cons(blue, cons(yellow, cons(grey, nil)))).$$

In the tree constraint domain, *Tree*, variables take finite trees as values. Constraints in the tree domain are equalities between *terms*, which are trees with some subtrees replaced by variables. Note that variables replace entire trees and not tree constructors.

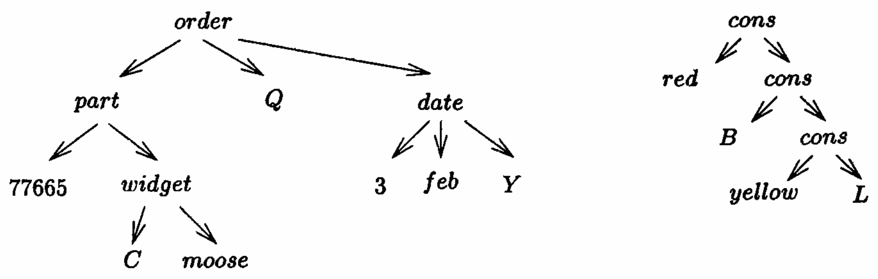

Figure 1.6 Two terms.

Definition 1.7

A *term* is defined recursively as follows: it is either a constant, or a variable, or a tree constructor together with an ordered list of $n \geq 1$ terms.

A *term equation* is of the form $s = t$ where s and t are terms. A valuation θ is a solution of term equation $s = t$ if $\theta(s)$ is syntactically identical to $\theta(t)$.

A *tree constraint* is a conjunction of term equations.

Two example terms are shown in Figure 1.6. The terms are written as

$$order(part(77665, widget(C, moose)), Q, date(3, feb, Y))$$

and

$$cons(red, cons(B, cons(yellow, L)))$$

respectively. In the tree constraint domain, the primitive constraints are term equations. Term equations are useful for constructing and accessing data stored in records or trees. The following constraint in effect builds the binary tree T containing "kim," "peter" and "harald" from the binary trees for "harald" and "peter":

$$K = node(H, kim, P) \land H = node(null, harald, null) \land P = node(null, peter, null).$$

It has a single solution

$$\{K \mapsto node(node(null, harald, null), kim, node(null, peter, null)),$$
$$H \mapsto node(null, harald, null),\ \ P \mapsto node(null, peter, null)\}.$$

Note that the terms K and $node(H, kim, P)$ under the valuation are both identical to

$$node(node(null, harald, null), kim, node(null, peter, null)),$$

and so the term equation is satisfied. Consider the constraint

$$Order = order(part(77665, widget(red, moose)), quantity(17), date(3, feb, 1994)) \land$$
$$Order = order(X, Y, date(Z, Month, U)).$$

The only solution ensures that the variable $Month$ takes the value feb, in some sense accessing the "month" field of the order. As a final example, the following constraint has the effect of extracting the first element H and the remainder T of the list represented by the right tree in Figure 1.4:

$$List = cons(red, cons(blue, cons(yellow, cons(grey, nil)))) \land List = cons(H, T).$$

The only solution to the constraint is

$$\{List \mapsto cons(red, cons(blue, cons(yellow, cons(grey, nil)))),$$
$$H \mapsto red,\ T \mapsto cons(blue, cons(yellow, cons(grey, nil)))\}.$$

In later chapters we shall see that tree constraints provide constraint logic programming languages with the full power of the usual recursive and non-recursive data structures used in traditional programming languages.

The most important question, of course, is how we determine whether a conjunction of term equations is satisfiable. The constraint solver for tree constraints described below is also a normalising solver that repeatedly eliminates variables from the term equations until a solved form is reached.

Algorithm 1.2: Tree constraint solver
INPUT: A conjunction of term equations.
OUTPUT: Returns *true* if they are satisfiable, otherwise *false*.
METHOD: The algorithm is shown in Figure 1.7. Cases are tried in order, so that the first condition that is satisfied by the equation will determine the case to be used. □

The algorithm works by selecting a term equation from the constraint to be solved, C, and either replacing it with an equivalent tree constraint or discovering unsatisfiability. If the term equation is of the form $x = t$ where x is a variable and t a term, then x is replaced everywhere by t and the equation is added to the solved form constraint S.

We can justify the correctness of the algorithm as follows. The constraint $C \land S$ has the same solutions after each iteration of the algorithm. For case (1), observe that any equation of the form $x = x$ is always satisfied whatever value is assigned to x, so it can be removed. For case (2), no tree which has root labelled f can ever be identical to a tree with root labelled g if the tree constructors are different ($f \not\equiv g$) or the number of children is different ($n \neq m$). For case (3), if two trees with the same tree constructor f and the same number of children n are to be equal then each of their corresponding subtrees must be equal. Case (4) simply swaps the two sides of the equation; any solution for $x = t$ is also a solution for $t = x$ and vice versa. Case (5) is a little more subtle to justify. Suppose there is a solution to the equation $x = t$. Then x is assigned a tree of, say, height h. Now t contains x but it is not just the variable x, so x must occur as an argument to some tree constructor. Thus t under the valuation is a tree of height $h+1$ or greater. But therefore it cannot

```
$x$ is a variable;
$s, t, s_i, t_j$ are terms;
$c$ is a term equation;
$C, C_0$ and $S$ are conjunctions of equations.


tree_solve($C$)
    if unify($C$) $\equiv$ $false$ then return $false$
    else return $true$
    endif

unify($C$)
    $S := true$
    while $C$ is not the empty conjunction
        let $C$ be of the form $c \wedge C_0$
        cases
        (1) $c$ is of the form $x = x$:
                $C := C_0$
        (2) $c$ is of the form $f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$
            where $f \not\equiv g$ or $n \neq m$:
            return $false$
        (3) $c$ is of the form $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$:
            $C := (s_1 = t_1 \wedge \cdots \wedge s_n = t_n) \wedge C_0$
        (4) $c$ is of the form $t = x$ where $t$ is not a variable:
            $C := (x = t) \wedge C_0$
        (5) $c$ is of the form $x = t$ and $t$ contains $x$:
            return $false$
        (6) $c$ is of the form $x = t$:
            substitute $t$ for $x$ throughout $C_0$ and $S$
            $C := C_0$
            $S := S \wedge c$
        endcases
    endwhile
    return $S$
```

Figure 1.7 A tree constraint solver.

be equal to the tree assigned to x, and so we have a contradiction. Hence there is no solution to $x = t$. Finally, case (6) is correct by the principle of substitutivity of equals for equals, since it simply incorporates the constraint information from c into C and S. Thus, we have proved the algorithm is correct. Proof of termination is left as an exercise.

Let us trace the execution of unify on a simple tree constraint

$$cons(Y, nil) = cons(X, Z) \wedge Y = cons(a, T).$$

The following table shows the values of C and S at the start of each iteration of the **while** loop in unify. We use underlining to show which primitive constraint gets processed in each iteration.

	C	S
Initially,	$\underline{cons(Y,nil) = cons(X,Z)} \land Y = cons(a,T)$	$true.$
Using rule (3),	$\underline{Y = X} \land nil = Z \land Y = cons(a,T)$	$true.$
Using rule (6),	$\underline{nil = Z} \land X = cons(a,T)$	$Y = X.$
Using rule (4),	$\underline{Z = nil} \land X = cons(a,T)$	$Y = X.$
Using rule (6),	$\underline{X = cons(a,T)}$	$Y = X \land Z = nil.$
Using rule (6),	$true$	$Y = cons(a,T) \land Z = nil \land X = cons(a,T).$

The resulting solved form is clearly satisfiable. By assigning any tree to T, for example nil, we can calculate assignments for each of the other variables that satisfy the equations. Thus

$$\{X \mapsto cons(a,nil), Z \mapsto nil, Y \mapsto cons(a,nil), T \mapsto nil\}$$

is a solution of the final constraint and therefore of the initial constraint. An example of an unsatisfiable constraint is $cons(X,Z) = cons(Z,nil) \land X = a$. The algorithm unify proceeds as follows:

	C	S
Initially	$\underline{cons(X,Z) = cons(Z,nil)} \land X = a$	$true,$
Using rule (3)	$\underline{X = Z} \land Z = nil \land X = a$	$true,$
Using rule (6)	$\underline{Z = nil} \land Z = a$	$X = Z,$
Using rule (6)	$\underline{nil = a}$	$X = nil \land Z = nil,$
$false$ using rule (2).		

Notice the strong similarity between unify and gauss_jordan. Both algorithms rewrite constraints to solved form. In addition, in both solvers the key step in obtaining the solved form is to eliminate a variable.

1.5 Other Constraint Domains

Numerous other constraint domains are of interest. Many of these have been closely studied in mathematics, for example Booleans, sets, sequences and groups. Here we take a look at Boolean constraints, sequence constraints and, as our last example of a constraint domain in this chapter, symbolic constraints for an application (planning) in artificial intelligence.

1.5.1 Boolean Constraints

Boolean primitive constraints are made up of Boolean variables, which may take the value $true$ (represented by 1) or $false$ (represented by 0), and Boolean operations such as conjunction (&), disjunction ($\vee$), exclusive-or ($\oplus$), implication ($\rightarrow$) and bi-implication ($\leftrightarrow$). Note that we deliberately use a different symbol for Boolean con-

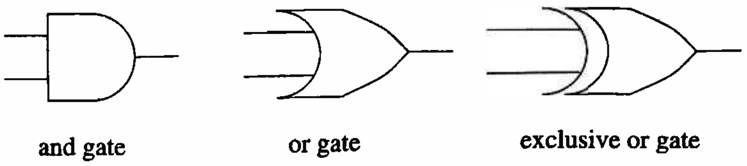

Figure 1.8 Gates for logic circuits.

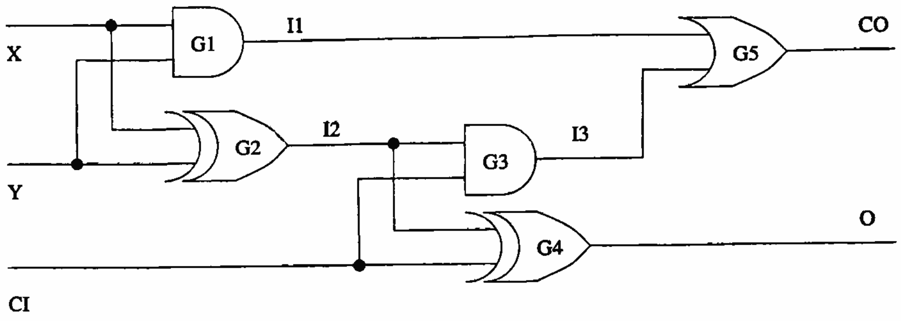

Figure 1.9 A full adder logic circuit.

junction (& as opposed to $\wedge$) to differentiate primitive constraints from constraints. The meaning of each of these functions on Booleans is given by the following truth tables, where, for example, $0 \rightarrow 1$ is 1 but $1 \rightarrow 0$ is 0:

$\neg$	
0	1
1	0

&	0	1
0	0	0
1	0	1

$\vee$	0	1
0	0	1
1	1	1

$\oplus$	0	1
0	0	1
1	1	0

$\rightarrow$	0	1
0	1	1
1	0	1

$\leftrightarrow$	0	1
0	1	0
1	0	1

One important application of Boolean constraints is for modelling logic circuits. A full adder is a circuit which takes three binary inputs X, Y and CI and gives two outputs O and CO. X and Y are the bits to be added, CI is the input carry bit, O is the output bit and CO is the output carry bit. The results of the full adder satisfy the arithmetic equation $O + 2 \times CO = X + Y + CI$. Different gate types are shown in Figure 1.8, and the circuit implementing a full adder is shown in Figure 1.9.

Each gate in the diagram is modelled by a Boolean primitive constraint as shown in the table below. The conjunction of these constraints models the entire circuit.

$I1 \leftrightarrow X \mathbin{\&} Y$	And gate $G1$,
$I2 \leftrightarrow X \oplus Y$	Xor gate $G2$,
$I3 \leftrightarrow I2 \mathbin{\&} CI$	And gate $G3$,
$O \leftrightarrow I2 \oplus CI$	Xor gate $G4$,
$CO \leftrightarrow I1 \vee I3$	Or gate $G5$.

For example, the above conjunction, conjoined with the information that $X \leftrightarrow 1 \wedge Y \leftrightarrow 0 \wedge CI \leftrightarrow 1$ has the single solution

$$\{X \mapsto 1, Y \mapsto 0, CI \mapsto 1, I1 \mapsto 0, I2 \mapsto 1, I3 \mapsto 1, O \mapsto 0, CO \mapsto 1\}.$$

This shows that the output of the circuit for input $X = 1$, $Y = 0$, and $CI = 1$ is $O = 0$ and $CO = 1$ as expected. In other words, $0 + 2 \times 1 = 1 + 0 + 1$.

A more interesting problem is to consider fault analysis of the adder. We associate with each gate Gi a variable F_i which is 1 if the gate is faulty and 0 if it is not. The behaviour of the adder is then given by the conjunction of the following primitive constraints:

$\neg F_1 \rightarrow (I1 \leftrightarrow X \;\&\; Y)$	And gate $G1$ works if not faulty,
$\neg F_2 \rightarrow (I2 \leftrightarrow X \oplus Y)$	Xor gate $G2$ works if not faulty,
$\neg F_3 \rightarrow (I3 \leftrightarrow I2 \;\&\; CI)$	And gate $G3$ works if not faulty,
$\neg F_4 \rightarrow (O \leftrightarrow I2 \oplus CI)$	Xor gate $G4$ works if not faulty,
$\neg F_5 \rightarrow (CO \leftrightarrow I1 \vee I3)$	Or gate $G5$ works if not faulty.

The following constraint enforces the reasonable hypothesis that at most one gate is faulty in the adder:

$$\neg(F_1\&F_2) \wedge \neg(F_1\&F_3) \wedge \neg(F_1\&F_4) \wedge \neg(F_1\&F_5) \wedge \neg(F_2\&F_3) \wedge$$
$$\neg(F_2\&F_4) \wedge \neg(F_2\&F_5) \wedge \neg(F_3\&F_4) \wedge \neg(F_3\&F_5) \wedge \neg(F_4\&F_5).$$

Suppose we conjoin this constraint information with some observed faulty behaviour of the gate. Say that, with inputs $X = 0$, $Y = 0$ and $CI = 1$, the outputs were $O = 0$ and $CO = 1$. Then, the only solution is

$$\{F_1 \mapsto 0, F_2 \mapsto 1, F_3 \mapsto 0, F_4 \mapsto 0, F_5 \mapsto 0\}.$$

So we know, if we assume that only one gate is faulty, it must be the second gate, $G2$.

There are many different approaches to solving Boolean constraints. Unfortunately the problem of deciding whether a Boolean constraint is satisfiable (SAT) is the most famous of the so-called NP-complete problems, a large and important class of intrinsically difficult problems. So far, all algorithms for solving SAT have a worst-case running time which is exponential in the size of the constraint. Moreover, it is impossible to do better, unless *every* NP-complete problem has a polynomial-time solution, something most computer scientists believe to be impossible. Nonetheless, here we give a polynomial-time solver for Boolean constraints!

```
C is a Boolean formula;
$\epsilon$ is a number between 0 and 1;
i, m and n are integers.

bool_solve(C)
    let m be the number of primitive constraints in C
    (assume $m > 1$)
    $n := \lceil \ln(\epsilon) / \ln(1 - (1 - 1/m)^m) \rceil$
    for i := 1 to n do
        generate a random valuation $\phi$ over the variables in C
        if $\phi$ satisfies C then return true endif
    endfor
    return unknown
```

Figure 1.10 Probabilistic Boolean solver.

Algorithm 1.3: Incomplete Boolean solver
INPUT: A Boolean constraint C, and ϵ, a number between 0 and 1, indicating the degree of incompleteness.
OUTPUT: *true* or *unknown*, indicating either that C is satisfiable or that C may or may not be satisfiable.
METHOD: The algorithm is shown in Figure 1.10. □

The price we pay for having a polynomial time algorithm is that the solver is not complete. That is, it sometimes returns the answer *unknown*. Later we shall see that solvers that do not always completely answer the satisfiability problem may still be useful.

The Boolean solver is an example of a ***probabilistic*** algorithm. One parameter of the algorithm is a number, ϵ, between 0 and 1, indicating the degree of incompleteness we are willing to tolerate. The larger the value of ϵ, the smaller the chance of a useful answer. The algorithm repeatedly guesses a random valuation, and, relying on an assumption that the input constraint is "random," utilises the fact that satisfiability of a random Boolean constraint is much more likely than unsatisfiability. Figure 1.11 shows some examples of n as a function of ϵ and m. Notice how the algorithm is particularly well suited for large constraints, since the number of iterations remains almost constant as the size grows, although, the larger the constraint, the more work is required to check that a valuation is a solution.

If the constraints given to this solver are in a particular form called ***conjunctive normal form*** (CNF), we can get a better measure of the incompleteness of the solver. In this case, the probability that the solver returns *unknown* is exactly ϵ.

A Boolean constraint is in conjunctive normal form if it is either 0 or 1 or a conjunction of primitive constraints each of the form $L_1 \vee \cdots \vee L_n$ where each L_i is of the form x or $\neg x$ and x is a Boolean variable. For example, the constraint $A \vee \neg B \wedge \neg A \vee B$ is in CNF. Any Boolean constraint can be written into an equivalent Boolean constraint in CNF (see Exercise 2.6a). The CNF constraint above is equivalent to $A \leftrightarrow B$.

m	5	50	500	5000	50000
ϵ					
0.1	6	6	6	6	6
0.01	12	11	11	11	11
0.001	18	16	16	16	16
0.0001	24	21	21	21	21

Figure 1.11 Values for n in terms of ϵ and m.

1.5.2 Sequence Constraints

Sequence constraints are another type of constraint. One important application of sequence constraints is in genetics. DNA, that is, deoxyribonucleic acid, is the material which forms the chromosomes of all known life. DNA consists of two long sequences of the molecules adenine (A), thymine (T), cytosine (C) and guanine (G), called *bases*, organized in a double helix shape. The bases in each strand are always paired together in the same way, adenine with thymine and cytosine with guanine. Thus, each strand uniquely determines the other.

Much research in genetics is devoted to taking a strand from an organism's DNA and finding the ordering of the bases in the strand. This is called *mapping* the DNA. Mapping allows a geneticist to decode the message contained in the organism's chromosomes. A large international effort (the Human GENOME project) is currently devoted to mapping and decoding the DNA of humans.

However, chromosomes are very long—for humans the chromosome's DNA strand may contain several billion bases. To map a chromosome, one of the DNA strands is broken into small manageable pieces which can be mapped individually. These pieces are known as *clones*, because once they have been extracted, they are inserted into a host chromosome and copied many times (or "cloned"). Different clones that are from the same section of chromosome are called *contigs* (from contiguous).

An important problem in genetics is, therefore, the reconstruction of an original chromosome from its contigs. Contigs may overlap, so the problem is to find the minimal length sequence which the contigs may yield when overlaid.

For example, imagine that we have the three contigs

```
A-A-A-A-T-C-G,
A-T-C-G-G-G-C,
G-C-C-A-T-T.
```

These can be combined to give the overall sequence `A-A-A-A-T-C-G-G-G-C-C-A-T-T` as follows:

```
A-A-A-A-T-C-G
      A-T-C-G-G-G-C
                G-C-C-A-T-T.
```

How the contigs can be overlaid to give this particular sequence is expressed by the sequence constraint:

$$\begin{aligned}
&T = UC_1 :: O_{12} :: UC_2 :: O_{23} :: UC_3 \land \\
&C_1 = UC_1 :: O_{12} \land C_2 = O_{12} :: UC_2 :: O_{23} \land C_3 = O_{23} :: UC_3 \land \\
&O_{12} \neq \epsilon \land O_{23} \neq \epsilon \land \\
&C_1 = a\text{-}a\text{-}a\text{-}a\text{-}t\text{-}c\text{-}g \land C_2 = a\text{-}t\text{-}c\text{-}g\text{-}g\text{-}g\text{-}c \land C_3 = g\text{-}c\text{-}c\text{-}a\text{-}t\text{-}t.
\end{aligned}$$

We use :: to denote sequence concatenation. T is the total or overall sequence, C_1, C_2 and C_3 are the contigs, UC_i is the non-overlapped portion of contig C_i and O_{ij} is the overlap between contigs C_i and C_j. As one would expect, the only solution to these constraints requires T to be *a-a-a-a-t-c-g-g-g-c-c-a-t-t*.

It may be possible to overlay the contigs in some other way. For instance, another possible overlapping is given by:

$$\begin{aligned}
&T = UC_3 :: O_{32} :: UC_2 :: O_{21} :: UC_1 \land \\
&C_3 = UC_3 :: O_{32} \land C_2 = O_{32} :: UC_2 :: O_{21} \land C_1 = O_{21} :: UC_1 \land \\
&O_{32} \neq \epsilon \land O_{21} \neq \epsilon \land \\
&C_1 = a\text{-}a\text{-}a\text{-}a\text{-}t\text{-}c\text{-}g \land C_2 = a\text{-}t\text{-}c\text{-}g\text{-}g\text{-}g\text{-}c \land C_3 = g\text{-}c\text{-}c\text{-}a\text{-}t\text{-}t.
\end{aligned}$$

However, in this case the constraints are unsatisfiable. There are another 4 ways we might overlay the contigs, corresponding to the other left-to-right orderings of the contigs, but in all cases these lead to unsatisfiable constraints.

1.5.3 Blocks World Constraints

Not all constraint domains are taken from mathematics. The ***blocks world*** is a well-studied artificial intelligence example of knowledge representation and planning. The blocks world (for our example) consists of a large floor space and a set of coloured objects: cubes, spheres and square pyramids.

A constraint domain for reasoning about the blocks world takes values from the set of available objects together with their position. For example, we might have two cubes, one yellow (yc) and one green (gc), a red sphere (rs), two pyramids, one green (gp) and one red (rp), as the set of possible objects. Primitive constraints on objects can force two references to be to the same object, for example $X = Y$, or restrict the colour, for example $red(X)$, or the shape, for example $sphere(Y)$. Others constrain the relative stacking position of the objects: $on(X, Y)$ requires that object X is on top of object Y, and $floor(X)$ requires that X is on the floor. These primitive constraints are allowed together with their negations, for example $X \neq Y$ (X and Y are different objects), $not\_green(X)$ (X is not green), and $not\_on(X, Y)$ (object X is not on Y).

The physical laws of the blocks world limit how objects may be placed. For example, an object must be on exactly one other object or the floor, a sphere cannot be placed on any object but the floor, and no object can be placed on a

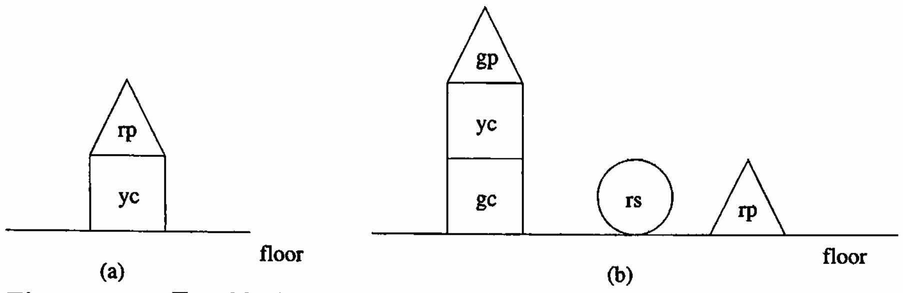

Figure 1.12 Two blocks world situations.

sphere or pyramid. A valuation (or situation) in the blocks world is a legitimate placement of each of the objects in the world. It assigns to each variable both an object and a position in the world. For simplicity, we will represent the position of the object simply by the object it is on.

The following constraint requires a red object to be on something:

$$red(X) \land on(X, Y).$$

A solution $\{X \mapsto rp\ on\ yc, Y \mapsto yc\ on\ floor\}$ is illustrated in Figure 1.12 (a). The following constraint requires a yellow object to be off the floor, a pyramid to be on an object which is not green, and two red objects to be on the floor:

$$yellow(X) \land on(X, Y) \land pyramid(Z) \land on(Z, T) \land not\_green(T) \land$$
$$red(U) \land red(V) \land floor(U) \land floor(V) \land U \neq V.$$

For our example world, the unique solution (apart from swapping U and V)

$$\{X \mapsto yc\ on\ gc, Y \mapsto gc\ on\ floor, Z \mapsto gp\ on\ yc, T \mapsto yc\ on\ gc,$$
$$U \mapsto rs\ on\ floor, V \mapsto rp\ on\ floor\}$$

is illustrated in Figure 1.12 (b).

1.6 Properties of Constraint Solvers

Except for bool_solve, all of the solvers we have examined so far are complete in the sense that they are able to determine for every constraint whether it is satisfiable (*true*) or unsatisfiable (*false*). In general, solvers may not have this property either because a complete algorithm is considered too expensive (as in the case of the Booleans) or a complete solver is unknown or cannot exist. For example, it is impossible to give an algorithm to determine whether a conjunction of nonlinear integer constraints has a solution. Thus, no complete solver for the domain of integers exists. In the case of nonlinear constraints over the reals, in theory it is decidable if a constraint is satisfiable or not, but the inherent complexity of doing so is highly exponential. Therefore, we cannot expect a complete solver to answer all satisfiability questions within a reasonable amount of time.

For these reasons we allow a solver to be *incomplete* and to answer some satisfaction questions with *unknown*, indicating that the solver cannot determine whether the constraint is satisfiable or unsatisfiable. Thus a solver can be formalised as follows:

Definition 1.8
A *constraint solver*, *solv*, for a constraint domain $\mathcal{D}$ takes as input any constraint C in $\mathcal{D}$ and returns *true*, *false* or *unknown*. Whenever $solv(C)$ returns *true* the constraint C must be satisfiable, and whenever $solv(C)$ returns *false* the constraint C must be unsatisfiable.

The constraint solver must answer the satisfiability question correctly or else return *unknown*. Note that the definition allows a constraint solver to be very weak. For example, the function which always returns *unknown* regardless of its input is a constraint solver for any domain (albeit not a very useful one).

Apart from being correct, there are a number of desirable properties we might expect our constraint solver to satisfy. The first relates to the textual form of the constraint. A constraint solver should give the same answers for a conjunction of primitive constraints regardless of their order, or the number of times a primitive constraint is repeated. In other words, the result of the constraint solver should depend only on the set of primitive constraints, $primitives(C)$, of the input constraint C.

The next desirable property a constraint solver should satisfy reflects our understanding of the meaning of conjunction. If C_1 is an unsatisfiable constraint, then any conjunction of C_1 with another constraint C_2 must also be unsatisfiable. A constraint solver should reflect this behaviour.

Finally, we would like the result of the constraint solver not to depend on the exact names of the variables, because the variables just represent place-holders. For example, $X \geq Y \wedge Y \geq 2$ and $U \geq V \wedge V \geq 2$ should not be distinguished. To define rigorously what we mean by a solver being independent of variable names, we introduce renamings and variants.

Definition 1.9
A *renaming* is a mapping from variables to variables in which no two variables are mapped to the same variable. We shall write a renaming ρ as the set

$$\{x_1 \mapsto y_1, \ldots x_n \mapsto y_n\}.$$

The result of *applying* ρ to a constraint C, written $\rho(C)$, is the constraint obtained by replacing each variable x in C by $\rho(x)$. A constraint C is a *variant* of constraint C' if there is a renaming ρ such that $\rho(C) \equiv C'$.

For example, $\rho_1 = \{X \mapsto U, Y \mapsto V\}$ is a renaming. $\rho_2 = \{X \mapsto Z, Y \mapsto Z\}$ is not a renaming since both X and Y are mapped to Z. Applying ρ_1 to $X \geq Y \wedge Y \geq 2$ gives $U \geq V \wedge V \geq 2$. Thus, $X \geq Y \wedge Y \geq 2$ is a variant of $U \geq V \wedge V \geq 2$ and vice versa.

Idealized constraint solvers almost always satisfy the set based, monotonic and variable name independent conditions which are defined below. In practice, however, some constraint solvers may occasionally treat constraints unfairly, that is, process some primitive constraints indefinitely while ignoring other primitive constraints. Such a constraint solver will not satisfy these conditions.

Definition 1.10
A constraint solver *solv* for constraint domain $\mathcal{D}$ is *well-behaved* if for any constraints C_1 and C_2 from $\mathcal{D}$:

set based: $solv(C_1) = solv(C_2)$ whenever $primitives(C_1) = primitives(C_2)$.

monotonic: If $solv(C_1) = false$ then $solv(C_1 \land C_2) = false$.

variable name independent: If C_1 is a variant of C_2 then $solv(C_1) = solv(C_2)$.

Except for bool_solve, the solvers we have examined so far are well-behaved. The reason that the Boolean solver is not well-behaved is that, because of its probabilistic nature, bool_solve(C) may not give the same answer even for the same constraint. For the remainder of this text, unless specifically noted otherwise, we shall assume that all solvers are well-behaved. As we have previously discussed, the strongest condition we can ask of a constraint solver, completeness, is not always achievable.

Definition 1.11
A constraint solver *solv* for constraint domain $\mathcal{D}$ is *complete* if for each constraint C in $\mathcal{D}$, $solv(C)$ is either $true$ or $false$. A solver which is not complete is said to be *incomplete*.

For example, gauss_jordan_solve is a complete constraint solver for the domain of linear real equations and tree_solve is a complete constraint solver for tree constraint domains. Clearly bool_solve is an incomplete solver. Complete solvers are the most desirable because they answer all questions. It is not too difficult to show that every complete solver is well-behaved. (See Exercise 1.10.)

1.7 (*) Determined Variables and Local Propagation

In previous sections we have seen a variety of constraint solvers. However, all of these were applicable to only one constraint domain. Here we present a general constraint solving algorithm that can be used with almost any type of constraints. This generality comes with a price—the solver is usually incomplete.

The algorithm is based on repeatedly detecting variables whose value is fixed by the constraint and eliminating these variables from the constraint. A core step in the algorithm is therefore determining variables which are fixed by a constraint to take a single value. We say such variables are determined by the constraint.

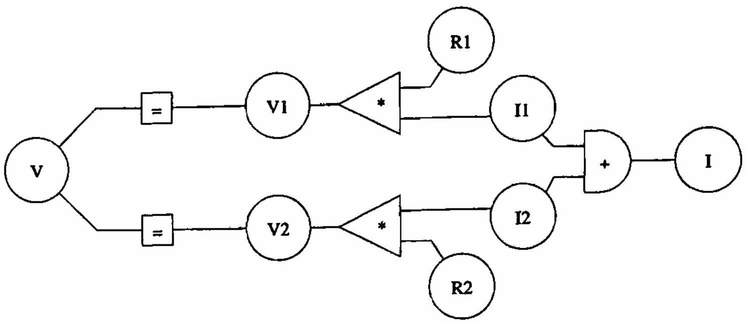

Figure 1.13 Network describing a simple electric circuit.

Definition 1.12
Let e be an expression containing no variables. A variable x is ***determined*** by C to have value e if every solution of C is also a solution of $x = e$.

For example, X is determined by $X = Y + 2Z - 2 \wedge Y = 1 - 2Z$ to have value -1 and X is determined by $2 = X \times X \wedge X \geq 0$ to have value $\sqrt{2}$.

Detecting determined values and simplifying the resulting constraints provides a general approach to solving constraints from any constraint domain. The idea arises from viewing constraints as a vehicle for "data-flow" propagation of values of variables. Consider the constraint describing the simple circuit shown in Figure 1.1:

$$V = V1 \wedge V = V2 \wedge V1 = I1 \times R1 \wedge V2 = I2 \times R2 \wedge I = I1 + I2.$$

This constraint can be represented by a network of operations shown in Figure 1.13. Each variable is shown in a circle and each operation is shown in another shape. An equality node, labelled with "=", ensures that the nodes it connects have the same value. Addition and multiplication nodes ensure that the result of applying the operation to the values of nodes connected to the incoming arcs (that is, those entering the flat side of the node) is equal to the value of the node on the outgoing arc (that is, the one leaving the non-flat side).

Now consider the additional constraint $V = 10 \wedge R1 = 5 \wedge R2 = 10$. This constraint corresponds to setting the values $V = 10$, $R1 = 5$ and $R2 = 10$. The corresponding network is shown at the top of Figure 1.14. Consider the operation of the network when these values are set. The value of the variable node for V can be propagated through the topmost equality node to set the value of $V1$ to 10. The resulting network is shown at the bottom of Figure 1.14. Then propagation through the topmost multiplication node can set $I1$ to 2 so as to satisfy $V1 = R1 \times I1$.

Propagation of the known values continues as follows. Propagation through the lower equality node sets $V2$ to 10, and then propagation through the other multiplication node sets $I2$ to 1. Finally, the addition node sets I to 3. At this point every variable has a determined value, and we can read the following solution from the network:

$$\{V \mapsto 10, R1 \mapsto 5, R2 \mapsto 10, V1 \mapsto 10, I1 \mapsto 2, V2 \mapsto 10, I2 \mapsto 1, I \mapsto 3\}.$$

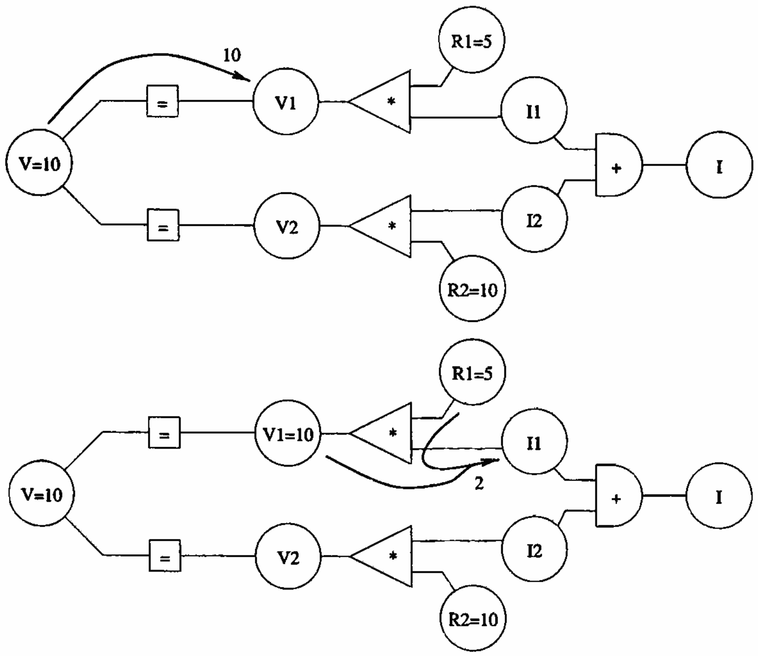

Figure 1.14 Propagation steps in the network.

The local propagation solver will return *true*.

As another example, imagine the additional constraint was instead $I1 = 2 \land V1 = 10$. In this case local propagation proceeds as follows. The topmost multiplication node determines that $R1 = 5$, the top equality node determines $V = 10$, and then the remaining equality node sets $V2 = 10$. No further information is available. Because not every variable is determined, local propagation cannot determine whether the constraint conjoined with the additional information $I1 = 2 \land V1 = 10$ is satisfiable or not, and will return *unknown*.

We now describe a general algorithm to perform local propagation. Local propagation can be thought of as manipulating two objects: a set of currently determined variables with their values and a constraint. We can represent the set of determined variables with their values using a solved form constraint containing these determined variables.

Definition 1.13
A constraint is in *determined solved form* if it has the form

$$x_1 = e_1 \land x_2 = e_2 \land \cdots \land x_n = e_n$$

where each x_i is a distinct variable and each e_i is a variable free expression.

The basic idea of local propagation is to repeatedly look at the primitive constraints in the constraint being solved and determine which other variables are constrained to take a single value, then to add this information to the determined solved form.

For example, if X is determined to be 3 then the constraint $X = Y + 2 \land Y + 1 + X = Z$ determines that Y must take value 1. Then with $X = 3 \land Y = 1$ the constraint determines that $Z = 5$.

Detecting when a variable is determined allows us to eliminate the variable. Unfortunately computing which variables are determined by an arbitrary constraint is at least as hard as determining whether the constraint is satisfiable. One key idea behind local propagation is that it is the interaction between arbitrary conjunctions of primitive constraints which is difficult to understand. In the execution of the local propagation algorithm, at each step we only determine which variables are determined by some single primitive constraint in the conjunction. Clearly this is much easier.

The generic algorithm relies on two domain specific functions, each of which takes a primitive constraint c. The first function, $satisfiable(c)$, is a solver for primitive constraints and returns *unknown*, *true* or *false*. Like all solvers, if it returns *true*, c must be satisfiable, and if it returns *false*, c must be unsatisfiable. The second function, $determines(c)$, takes a primitive constraint and returns a determined solved form,

$$x_1 = e_1 \land x_2 = e_2 \land \cdots \land x_n = e_n,$$

where for each i, c determines that x_i takes value e_i.

The local propagation algorithm works by repeatedly selecting a primitive constraint. If the primitive constraint is definitely unsatisfiable, the algorithm returns *false* since the original constraint is also unsatisfiable. Otherwise, the algorithm finds which variables this primitive constraint determines, eliminates them from the constraint and adds them to the determined solved form. If, at any stage, a constraint contains no variables and is satisfiable, it can be removed. This process continues until no further step can be taken. If the final constraint is *true*, the original system is definitely satisfiable. Otherwise the algorithm returns *unknown*.

Algorithm 1.4: Local propagation solver
INPUT: A constraint C.
OUTPUT: Returns *true* if C is shown to be satisfiable, *false* if C is shown to be unsatisfiable, otherwise *unknown*.
METHOD: The algorithm is shown in Figure 1.15. □

As an example of the execution of local_propagation_solve, consider the Boolean constraint

$$X \leftrightarrow Y \land \neg Y \land \neg Y \rightarrow X.$$

Initially C is this constraint and S is *true*. C_0 is set to C, and each primitive constraint in C_0 is examined in turn. We assume that the primitive constraints are processed in textual order. The first constraint selected is $X \leftrightarrow Y$, so c is set to this and C_0 is set to $\neg Y \land \neg Y \rightarrow X$. Processing of $X \leftrightarrow Y$ leaves S and C unchanged

```
x is a variable;
e is an expression not involving variables;
c is a primitive constraint;
C, C_0, C_1 are constraints;
S is a constraint in determined solved form.

local_propagation_solve(C)
  S := true
  repeat
    C_0 := C
    while C_0 is not the empty conjunction
      let C_0 be of the form c ∧ C_1
      C_0 := C_1
      if satisfiable(c) ≡ false then
        return false
      elseif satisfiable(c) ≡ true and vars(c) ≡ ∅ then
        C := C with primitive constraint c removed
      endif
      S := S ∧ determines(c)
      for each constraint (x = e) ∈ determines(c) do
        replace x by e in C and C_0
      endfor
    endwhile
  until S and C are unchanged
  if C is the empty conjunction then
    return true
  else
    return unknown
  endif
```

Figure 1.15 Local propagation solver.

as $satisfiable(X \leftrightarrow Y)$ returns *true* but $X \leftrightarrow Y$ does not determine any variables. Next c is set to $\neg Y$ and C_0 becomes $\neg Y \rightarrow X$. Now, $\neg Y$ is satisfiable and also determines that Y is assigned 0. Thus S becomes $Y = 0$, C becomes

$$X \leftrightarrow 0 \wedge \neg 0 \wedge \neg 0 \rightarrow X$$

and C_0 becomes $\neg 0 \rightarrow X$ as Y is replaced by 0. Next $\neg 0 \rightarrow X$ is processed. This is satisfiable and determines that X is assigned 1. Thus S becomes $Y = 0 \wedge X = 1$ and C becomes

$$1 \leftrightarrow 0 \wedge \neg 0 \wedge \neg 0 \rightarrow 1.$$

The iteration finished because all primitive constraints have been processed. In the next iteration of the **repeat** loop, C_0 is set to $1 \leftrightarrow 0 \wedge \neg 0 \wedge \neg 0 \rightarrow 1$ and then $1 \leftrightarrow 0$ is processed. This is unsatisfiable, so the algorithm returns *false*, indicating that the original constraint is unsatisfiable.

As another example of the algorithm's operation, again consider the simple circuit show in Figure 1.1 together with specific values for the voltage of the battery and the resistance of the resistors. The initial constraint C describing it is

$$V = V1 \land V = V2 \land V1 = I1 \times R1 \land V2 = I2 \times R2 \land I = I1 + I2 \land$$
$$V = 10 \land R1 = 5 \land R2 = 10.$$

In the first iteration of the **repeat** loop, if primitive constraints are processed in textual order, processing of the first five primitive constraints

$$V = V1,\ V = V2,\ V1 = I1 \times R1,\ V2 = I2 \times R2,\ I = I1 + I2$$

leaves S and C unchanged as each is satisfiable and does not determine the value of a variable. Processing of

$$V = 10 \land R1 = 5 \land R2 = 10$$

leads to the elimination of V, $R1$ and $R2$. Thus S becomes

$$V = 10 \land R1 = 5 \land R2 = 10$$

and C becomes

$$10 = V1 \land 10 = V2 \land V1 = I1 \times 5 \land V2 = I2 \times 10 \land I = I1 + I2 \land 10 = 10 \land 5 = 5 \land 10 = 10.$$

In the next iteration, processing of $10 = V1$ and $10 = V2$ causes S to become

$$V = 10 \land R1 = 5 \land R2 = 10 \land V1 = 10 \land V2 = 10$$

and C becomes

$$10 = 10 \land 10 = 10 \land 10 = I1 \times 5 \land 10 = I2 \times 10 \land I = I1 + I2 \land 10 = 10 \land 5 = 5 \land 10 = 10.$$

Now processing of $10 = I1 \times 5$ and $10 = I2 \times 10$ leads to S becoming

$$V = 10 \land R1 = 5 \land R2 = 10 \land V1 = 10 \land V2 = 10 \land I1 = 2 \land I2 = 1$$

and C becomes

$$10 = 10 \land 10 = 10 \land 10 = 2 \times 5 \land 10 = 1 \times 10 \land I = 2 + 1 \land 10 = 10 \land 5 = 5 \land 10 = 10.$$

Processing of $I = 2 + 1$ eliminates I and causes S to become

$$V = 10 \land R1 = 5 \land R2 = 10 \land V1 = 10 \land V2 = 10 \land I1 = 2 \land I2 = 1 \land I = 3$$

and C becomes

$$10 = 10 \land 10 = 10 \land 10 = 2 \times 5 \land 10 = 1 \times 10 \land 3 = 2 + 1 \land 10 = 10 \land 5 = 5 \land 10 = 10.$$

Subsequent processing will remove all of the primitive constraints in C. Finally, the algorithm will return *true* since each primitive constraint is satisfiable and contains no variables.

Local propagation solvers are simple to implement and quite good at handling some kinds of constraints. However, they are usually not complete. One reason is that they are weak at solving constraints with cyclic dependencies. For example, consider the constraint

$$X = Y + Z \land X = T - Y \land T = 5 \land Z = 7.$$

A local propagation solver will detect the determined variables $T = 5$ and $Z = 7$. However the constraint determines the variables X and Y to be 6 and -1 respectively. This will not be found by local propagation because it cannot be determined by looking at the primitive constraints one at a time. Local propagation behaves as above for the unsatisfiable constraint

$$X = Y + Z \land X = T - Y \land T = 5 \land Z = 7 \land Y \geq 2,$$

determining that $T = 5$ and $Z = 7$ but not that $X = 6$ and $Y = -1$.

Another weakness of local propagation solvers is their handling of non-equality primitive constraints. Usually this is very weak because the *determines*(c) relation for non-equality primitive constraints c is almost always empty.

In Chapter 3 we will investigate more powerful propagation based approaches to constraint solving, which are used to solve constraints over finite domains.

1.8 Summary

Constraints are powerful tools for modelling many real-world problems. In this book, a constraint is considered to be a conjunction of primitive constraints. A constraint can be understood only in the context of a constraint domain, such as Boolean values, trees or real numbers, which dictates the meaning of the function and relation symbols and the values that variables in the constraint can range over.

One of the most fundamental questions we can ask about a constraint is whether it is satisfiable, and, if so, what a solution looks like. A constraint solver answers such questions. Ideally, a constraint solver is complete, that is always answer *true* or *false*, but a constraint solver may also be incomplete, sometimes returning *unknown*. Even incomplete solvers should be well-behaved in the sense that they should be monotonic, set-based and variable name independent.

In this chapter we gave examples of complete constraint solvers for linear real equality constraints, finite-tree constraints, and an incomplete solver for Boolean constraints. We also examined some other constraint domains: the blocks worlds and sequence constraints. Finally we presented local propagation, a general approach to constraint solving for any domain.

1.9 Exercises

1.1. Model an electric circuit similar to the one in Figure 1.1, except the resistors R_1 and R_2 should be serially connected. Investigate the behaviour of the resulting circuit under the constraints given in Section 1.2.

1.2. Determine the result of applying gauss_jordan to the following constraints:

(a) $X = 3 \wedge Y = 2 - X \wedge Z = 3 + X + Y$,

(b) $Y = 2 - X \wedge Z = 3 + X + Y \wedge Z = 4$.

1.3. Give a term equation which will force the use of rules (1) and (5) in unify.

1.4. Using unify, determine which of the following term equations are satisfiable:

(a) $p(X,X,Y) = p(f(U),Y,f(V))$,

(b) $p(X,X,Y) = p(f(U),Y,a)$,

(c) $p(f(X,Y),g(Y),X) = p(U,g(g(V)),a)$,

(d) $p(a,X,f(g(Y))) = p(Z,h(Z,W),f(W))$,

(e) $p(f(X,Y),f(X,Z),Z) = p(f(U,a),f(V,V),f(U,a))$.

1.5. (*) A common way to prove that an algorithm terminates for all inputs is to come up with a ***bound function*** for one or more of the variables. A bound function returns non-negative integers. If one can show that the bound function applied to (some of) the algorithm's variables yields ***strictly decreasing*** values in successive iterations, then the algorithm must eventually halt.
The objective here is to argue that unify terminates for all input. In a first attempt to give a bound function, we may choose to let the function return the number of symbols in the constraint C. But unfortunately, not all rules in unify decrease the total number of symbols in the equation set. For example, application of rule (6) typically *increases* it!
Show that unify nevertheless must terminate. [Hint: There is something (in a sense more important) which rule (6) *does* decrease.]

1.6. Try out bool_solve in an attempt to show that

$$(\neg U \vee V) \wedge (U \vee \neg V) \wedge (X \vee Y) \wedge (Y \vee Z) \wedge (Z \vee X)$$

is satisfiable. Choose ϵ to be 0.1 and generate random truth assignments ϕ by throwing a coin (tail = 1, head = 0), once per variable.

1.7. Huey, Dewey and Louie are being questioned by their uncle. These are the statements they make:

Huey: "Dewey and Louie had equal share in it; if one is guilty, so is the other."

Dewey: "If Huey is guilty, then so am I."

Louie: "Dewey and I are not both guilty."

Their uncle, knowing that they are cub scouts, realises that they cannot tell a lie. Has he got sufficient information to decide who (if any) are guilty? Model the problem as Boolean constraints, and decide.

1.8. In the context of the example blocks world used in Section 1.5, count the number of solutions to the following blocks world problems:

(a) $red(X) \land floor(Y) \land on(Y, X)$,

(b) $green(X) \land on(X, Y) \land floor(Y)$,

(c) $green(X) \land pyramid(Y) \land yellow(Z) \land on(X, Z)$.

1.9. (*) Give an algorithm for a (possibly incomplete) solver for a constraint domain of sets. The only constant of the domain is $\emptyset$ representing the empty set. The constraint relations are $\subseteq, \subset, \neq, =$ with the usual meaning. Although always returning *unknown* is correct, try to make your constraint solver as strong as possible (that is, answering *true* or *false* in as many cases as possible).

1.10. (*) Prove the following: if *solv* is a complete constraint solver, then *solv* is well-behaved.

1.11. (*) Determine the answers for local_propagation_solve applied to the problems in Question 1.2.

1.12. (*) Use local propagation to determine the satisfiability of $X = 2Y \land Y = 1 \land Z = X \times X \land Y \leq Z$.

1.13. (*) Find conditions that ensure that local_propagation_solve is well-behaved.

1.10 Practical Exercises

This book is designed for use with real constraint logic programming languages. Many of these are available free for research and educational purposes. One widely used constraint programming language is $CLP(\mathcal{R})$, a constraint programming language for a constraint domain that combines tree constraints with real arithmetic constraints using a floating point representation of real numbers. It is available for most architectures including the majority of Unix architectures, Linux, DOS and OS/2. It can be obtained by sending an email request to Joxan Jaffar (`joxan@iscs.nus.sg`).

$CLP(\mathcal{R})$ includes a constraint solver which handles real arithmetic constraints and tree constraints. $CLP(\mathcal{R})$ is usually executed by typing the command `clpr` from the command shell. The $CLP(\mathcal{R})$ system presents a prompt to the user.

```
1 ?-
```

Constraints can be typed in at the prompt and $CLP(\mathcal{R})$ will invoke its constraint solver, and give one of three answers: `*** Yes` corresponding to *true*, `*** No` corresponding to *false*, and `*** Maybe` corresponding to *unknown*. If the result is not `*** No` the system also returns a representation of the constraints (an equivalent constraint to that which was typed in).

Constraints are represented using the following conventions: `*` is used for multiplication, linear expressions need explicit multiplication symbols, that is $2Y$ must be written as `2 * Y`, conjunction ($\wedge$) is represented using a comma "`,`" $\geq$ and $\leq$ are represented by "`>=`" and "`<=`" respectively, and finally a full stop (or period) "`.`" is used to terminate the constraint. To interrupt execution the user can type *control-C*, that is type "c" while holding down the control key. To quit from $CLP(\mathcal{R})$ the user can type *control-D*, that is type "d" while holding down the control key.

For example, the constraint $1 + X = 2Y + Z \wedge Z - X = 3$ is typed as shown in the following *emphasized text* after the prompt

```
1 ?- 1 + X = 2 * Y + Z, Z - X = 3.
```

This results in the system returning the answer

```
Y = -1
X = Z - 3

*** Yes
```

The last line `*** Yes` indicates that the constraint solver has determined that the constraints are satisfiable. Similarly, typing the following *emphasized text*

```
2 ?- 1 + X = 2 * Y + Z, Z - X = 3, X + Y = 5 + Z.
```

results in the system returning the answer

```
*** No
```

indicating that the constraint solver has determined the constraints are unsatisfiable. Finally, typing the following constraint

```
3 ?- X * X <= -1.
```

results in the system returning the answer

```
X * X <= -1

*** Maybe
```

indicating the solver cannot determine satisfiability or unsatisfiability.

Term equations are typed exactly as they appear in the text. For example

```
4 ?- cons(Y,nil) = cons(X,Z), Y = cons(a,T).
```

results in the system returning

```
X = Y
Z = nil
Y = cons(a, T)

*** Yes
```

indicating the constraint is satisfiable.

Prolog is a constraint logic programming language over the domain of tree constraints. However many recent Prolog systems include other constraint solving facilities. SICStus Prolog release 3 includes libraries for real arithmetic constraint solving. It provides solvers which use two different representations of numbers: infinite precision rational numbers, and floating point numbers. SICStus is widely used and available from the Swedish Institute of Computer Science (see `http://www.sics.se/isl/sicstus.html`).

SICStus Prolog is usually executed by typing the command `sicstus` from the command shell. The system presents a prompt to the user.

```
| ?-
```

In order to use the arithmetic constraint solving facilities one needs to load one of the constraint solving libraries either by typing:

```
| ?- use_module(library(clpq)).
```

to load the rational number solver, or

```
| ?- use_module(library(clpr)).
```

to load the floating point solver.

Constraints can be typed in at the prompt and SICStus Prolog will use the loaded constraint solver to determine satisfiability of the constraint. It will either return no if the solver determines the constraint to be unsatisfiable or return a representation of the constraints which is equivalent to the typed constraint. Both arithmetic solvers are incomplete. If the solver cannot determine whether the constraint is satisfiable or not, the answer returned by the system starts with a term of the form `nonlin:`t, indicating the result *unknown*.

Constraints are represented using almost the same notational conventions as $CLP(\mathcal{R})$ above, except $\leq$ is written as "=<" rather than "<=", a $\neq$ primitive constraint is supported written as "=/=", and importantly all primitive constraints or conjunctions of primitive constraints must be enclosed between braces "{" and "}". Term equations are typed exactly as they appear in the text, as for $CLP(\mathcal{R})$. Interrupting execution and quitting from SICStus Prolog is the same as for $CLP(\mathcal{R})$.

For example, using the rational constraint solver the constraint $1 + X = 2Y + Z \land Z - X = 3$ is typed as shown in the following *emphasized text* after the prompt

```
    | ?- {1 + X = 2 * Y + Z, Z - X = 3}.
```

This results in the system returning the answer

```
    Y = -1,
    {Z = 3+X} ?
```

The question mark indicates the system is waiting for input, for the purposes of this chapter just type Enter or Return in response. The answer yes will then be displayed. Since the answer does not begin with `nonlin:` this indicates the constraint solver has determined that the constraint is satisfiable.

If the floating point constraint solver is used the answer is slightly different because the numbers are represented differently. For the constraint above the system returns the answer

```
Y = -1.0,
{Z = 3.0+X} ?
```

Using the rational constraint solver typing the following *emphasized text*

```
| ?- {1 + X = 2 * Y + Z, Z - X = 3, X + Y = 5 + Z}.
```

results in the system returning the answer

```
no
```

indicating that the constraint solver has determined the constraint is unsatisfiable. Finally, typing the following constraint

```
| ?- {X * X =< -1}.
```

results in the system returning the answer

```
nonlin:{1+X^2=<0} ?
```

indicating that the solver cannot determine satisfiability or unsatisfiability.

P1.1. Use $CLP(\mathcal{R})$ or SICStus Prolog to determine the satisfiability of the constraints

(a) $X = 3 \land Y = 2 - X \land Z = 3 + X + Y$,

(b) $Y = 2 - X \land Z = 3 + X + Y \land Z = 4$.

Compare the constraint output by the system with your answer to Question 1.2.

P1.2. Write down what you expect as an answer to the following two constraints, then try them using $CLP(\mathcal{R})$ or SICStus Prolog.

(a) $X = 3 \land Y = 0 \land Z = X/Y$,

(b) $X = 0 \land Y = 0 \land Z = X/Y$.

P1.3. Use $CLP(\mathcal{R})$ or SICStus Prolog to determine the satisfiability of the constraints.

(a) $p(X, X, Y) = p(f(U), Y, f(V))$

(b) $p(X, X, Y) = p(f(U), Y, a)$

(c) $p(f(X, Y), g(Y), X) = p(U, g(g(V)), a)$

(d) $p(a, X, f(g(Y))) = p(Z, h(Z, W), f(W))$

Compare the constraint output by the system with your answer to Question 1.4.

P1.4. Use $CLP(\mathcal{R})$ or SICStus Prolog to determine the voltage V and current I in the circuit shown in Figure 1.1 when the resistor values are $R_1 = 4$ and $R_2 = 3$ and the current $I_1 = 3$.

P1.5. Write a constraint that describes the house building project of Figure 1.2 with the additional constraint that stages B and D of the project are reached at the same time. Use $CLP(\mathcal{R})$ or SICStus Prolog to determine the earliest completion time of the house with this new constraint. [Hint: Type in the constraints and see what the system says.]

P1.6. (*) Try the term constraint $X = f(X)$ in $CLP(\mathcal{R})$ or SICStus Prolog. How does the system respond? Try the missing constraint from Question 1.4, that is $p(f(X,Y), f(X,Z), Z) = p(f(U,a), f(V,V), f(U,a))$. Execute unify by hand on the constraint $X = f(X)$ but ignoring case (5) (thus using (6)). What does unify answer? Now imagine displaying the tree which results. For efficiency the $CLP(\mathcal{R})$ system does not perform the check in case (5). Can you explain its behaviour on these two constraints?

P1.7. (*) Sometimes the $CLP(\mathcal{R})$ or SICStus Prolog system answers `*** Maybe` or `nonlin:`*t* indicating an *unknown* answer. Try the following constraints:

(a) $X \times Y = Z$,

(b) $X \times Y = Z \wedge Z = 2$,

(c) $X \times Y = Z \wedge Y = 2$,

(d) $X \times Y = Z \wedge X = 2$,

(e) $X \times Y = Z \wedge Y = 0$.

Can you give a rule for determining when the solver answers *unknown*?

1.11 Notes

The concept of constraint solving has a long history in mathematics. Finding solutions to equations over the integers is one of the oldest mathematical problems and the Babylonians are known to have solved a variety of equations with two variables. Several Greek mathematicians studied equation solving, most notably Diophantus, who wrote his *Arithmetica* around the year 250. Diophantus solved many specific equational problems, but it is unlikely that he or any of his contemporaries knew and applied general solving methods. According to Struik [127], the first general solution in integers to equations of the form $aX + bY = c$ is due to the Indian mathematician Brahmagupta from the seventh century. (Such equations are today referred to as "Diophantine" although Diophantus, unlike Brahmagupta, allowed rational solutions and disallowed negative solutions.) In the ninth century, the Persian astronomer al-Khwarizmi wrote an influential treatise on equation solving, ***Hisab al jabr wal muqabala.*** The term "algebra" originated from the title which, until the nineteenth century, meant nothing but "equational reasoning." Incidentally, the term "algorithm" is also derived from al-Khwarizmi's latinised name.

Until the sixteenth century, constraint solving was dealt with only through cumbersome verbal descriptions of problems and methods. The use of symbols such as '+', '-', and '=' (or symbols resembling those) made reasoning much easier, and the next step forward was taken by Viète who, in the late sixteenth century, began to use letters for constants and variables. This allowed many practical and theoretical breakthroughs, including Descartes's application of algebra to problems in geometry.

By the end of the eighteenth century, solving linear equations by variable elimination was a standard technique, even though a precise algorithm had not been published. Gauss used variants (often called *Gaussian elimination*) of the Gauss-Jordan elimination method as early as 1809, but the method given in Section 1.3 is usually attributed to an 1829 paper by Gauss [54]. However, Schrijver [116] suggests that the Gauss-Jordan elimination method was not spelled out in detail before Clasen did so in 1888.

The algebra of propositional logic is due to Boole [13]. Many methods for solving Boolean constraints are known. The probabilistic algorithm used for the Boolean solver in Section 1.5.1 is related to one suggested by Wu and Tang [145]. For other automated approaches to Boolean constraint solving see [22, 93, 9].

In the late twentieth century, Frege extended Boole's work by developing what would, later, be known as predicate logic. Terms, or trees, in the sense of Section 1.4, and tree constraint solving is the cornerstone of most automated deduction. The tree constraint solver given in Section 1.4 is essentially due to Herbrand [65]. Independently, a rather different algorithm was given by Robinson, as part of the so-called resolution approach to automated theorem proving [112].

Apart from mathematics, the two areas that have had the greatest impact on a modern theory of constraints and their use in automated problem solving are Operations Research (OR) and Artificial Intelligence (AI). OR is concerned with building mathematical *models* of real world situations in order to allow the experimental analysis of problems, typically by computer. Much of the motivation for building computers came from OR and its wartime applications. Since most real-world problems include some notion of "constraints," many of the techniques developed in OR (linear programming, integer programming and so on) involve constraint solving.

AI is an umbrella term for a number of automated problem-solving techniques, which goes back to the sixties. We shall have more to say about the influence of AI after we have discussed other constraint operations in the next chapter. The blocks world was suggested by Sacerdoti [114] as an interesting "plan-formation" problem. For an overview of the state-of-the-art in constraint-based diagnosis, planning, scheduling, language understanding, machine vision, and other areas, see Freuder and Mackworth [50].

Sequences have a long history in computer science and are more commonly referred to as strings. The overlaying of contigs can be seen as a slightly more complicated instance of the standard problem of string matching, that is looking for one instance of a string in another. Well known algorithms for string matching like Knuth-Morris-Pratt [80] and Boyer-Moore [17] can be adapted to solve the problem. Solving more complicated constraints over sequences is considerably more difficult. The first complete constraint solver for sequence constraints is due to Makanin [91], while a more efficient solver results from the work of Jaffar [68].

Local propagation has been developed independently by a number of researchers to solve arithmetic and Boolean constraints. The term is generally attributed to Sussman and Steele [132], who used local propagation to solve constraints in the early constraint language CONSTRAINTS. Sutherland used it even earlier in SKETCHPAD [133] where it was called the "One Pass Method." Roth [113] used local propagation to solve satisfiability of Boolean constraints for test pattern generation for digital circuits.