12 Other Constraint Programming Languages

In the previous chapter we examined constraint databases which we viewed as CLP programs but with a different evaluation mechanism. However, constraint logic programming languages are not the only approach to constraint programming. In this chapter we will investigate other constraint programming languages and paradigms and compare them with the CLP paradigm.

We first look at concurrent constraint programming languages which extend CLP languages by providing asynchronous communication between "agents" by using constraint entailment. We also look at constraint handling rules which are a useful variant of concurrent constraint programming languages specifically designed for writing constraint solvers. Next we look at various attempts to combine constraints with other programming paradigms. This research is motivated by the view that CLP is "constraints + logic programming" and so we have "constraints + functional programming", "constraints + term rewriting" and "constraints + imperative programming." Then we examine "constraint solving toolkits" which are libraries for traditional programming languages that provide many of the features of CLP. Finally, we look at mathematical languages that make use of constraints—modelling languages for operations research and symbolic algebra packages.

We shall see that many of the other approaches extend or are based on the CLP paradigm. This is not surprising since, in a very real sense, CLP languages are the archetypal constraint programming language. One reason is historical—they were one of the first true constraint programming languages, so have had great influence on the design and implementation of subsequent languages. A second reason is that the constraint logic programming paradigm is at the heart of any other constraint programming language because, in essence, constraint programs are simply user-defined constraints and any reasonable constraint programming language will provide user-defined constraints.

12.1 The CLP Paradigm

Before we examine the other approaches to constraint programming, it is worthwhile summarizing the key features of the CLP paradigm, since this will be the yardstick with which we shall compare the other approaches.

As we saw in Part 2, CLP programs consist of rules which define user-defined constraints or predicates. Rules may be recursive and there may be multiple rules in the definition of a predicate. Multiple rules allow the programmer to express choice and mean that a goal may have more than answer. The default behaviour of the CLP system is to find answers, one at a time, to the given goal. However, the programmer may also specify that they want the answer which minimizes a given objective function.

Programs are parametric in the choice of the underlying constraint domain, solver and simplifier. Different CLP languages and systems result from different choices. However, they all share the same underlying evaluation mechanism—a depth-first left-to-right search through the goal's derivation tree. In this book we have examined three CLP languages in some detail: $CLP(Tree)$ which provides tree constraints and may be considered the core of the logic programming language Prolog, $CLP(\mathcal{R})$ which extends $CLP(Tree)$ by providing arithmetic constraints over the real numbers and $CLP(FD)$ which extends $CLP(Tree)$ by providing constraints over finite domains.

CLP languages can be understood from two complementary viewpoints. The first is that they are modelling languages. From this viewpoint, the job of the CLP programmer is to process the constraints which model the problem at hand into a form which is suitable for the underlying constraint solver. Term constraints and recursion mean that the modelling language is very powerful, allowing the model to make use of complex data structures and iteration.

The second viewpoint is that they are programming languages. Because of tree constraints and recursion they are computationally adequate, so in principle they can be used to implement any desired algorithm. This viewpoint better describes more advanced CLP programming activities, such as extending the underlying constraint solver or programming different types of search, for instance, by developing a problem specific labelling predicate.

CLP languages are, of course, quite different to traditional programming languages. One major difference is that search is built into the CLP evaluation mechanism. The system tries each rule until it finds an answer—a priori it doesn't know which rule to use, so must try all of them. This is often called ***don't know non-determinism.***

Apart from search, however, evaluation of a CLP program is similar to the evaluation of a program written in a procedural language. Predicates behave much like procedures—procedures are executed by evaluating the statements in the procedure definition in order, while predicates are evaluated by evaluating the literals in their definition proceeding left-to-right. If the literal is a constraint, it is added to the constraint store, and, if it is a call to a predicate, the call is evaluated.

12.2 Concurrent Constraint Programming Languages

We now examine other approaches to constraint programming. We shall start with concurrent constraint programming, since this is firmly based on the CLP paradigm.

Extending constraint logic programming with dynamic scheduling, as discussed in Section 9.6, not only allows us to write programs which work in a variety of modes of usage. It also allows a completely different style of programming in which we have "processes" or "agents" which execute concurrently and which communicate through the global constraint store.

Example 12.1

Consider the following program. The first two rules define a producer prod which produces a list of a's and the remaining rules define a consumer con which consumes the list of a's as they are produced. Note that the consumer predicate con delays until the list it is operating upon is constructed.

```
prod([]).                  (P1)
prod([a|L]) :- prod(L).    (P2)

:- delay_until(nonvar(L),con(L)).
con([]).                   (C1)
con([a|L]) :- con(L).      (C2)
```

The goal `con(L), prod(L)` has an infinite number of successful derivations since `prod(L)` produces lists of arbitrary length. If the leftmost enabled literal is always selected for rewriting, each derivation has essentially the same form. As soon as a rule for a prod literal is used to build more of the list L, the consumer con will "consume" the new element. Thus there is a form of synchronisation between the consumer and producer. An example simplified derivation is

$$\begin{array}{c}
\langle con(L), \underline{prod(L)} \mid true\rangle \\
\Downarrow P2 \\
\langle \underline{con(L)}, prod(L') \mid L = [a|L']\rangle \\
\Downarrow C2 \\
\langle con(L'), \underline{prod(L')} \mid L = [a|L']\rangle \\
\Downarrow P1 \\
\langle \underline{con(L')} \mid L = [a] \land L' = []\rangle \\
\Downarrow C1 \\
\langle \square \mid L = [a]\rangle.
\end{array}$$

The answer is $L = [a]$, but the important feature of the derivation is how the consumer only executes after each part of the list is produced.

Concurrent constraint programming (CCP) languages are based on this "process" reading of CLP rules in which a user-defined constraint is viewed as a *process* and a state is regarded as a network of processes linked through shared variables by means of the store. Processes communicate by adding constraints to the store and synchronize by waiting for the store to enable a delay condition.

Rules in a CCP language have the following form

$$p(X_1, \ldots, X_m) \text{ :- } G \mid L_1, \ldots, L_n$$

where the literal $p(X_1, \ldots, X_m)$ is the process defined by the rule, the literals $L_1, \ldots, L_n$ form the *body* of the rule and G is the *guard* which is a delay condition that must be enabled for the rule to be used. The syntax we use follows that of early CCP languages—modern CCP languages usually use a process notation which although semantically equivalent to the early notation, obscures the strong relationship with CLP languages.

Thus the above program might be written as the CCP program:

```
prod(X) :- true | X=[].
prod(X) :- true | X = [a|L], prod(L).

con(Y) :- emptylist(Y) | true.
con(Y) :- nonemptylist(Y) | Y=[a|L], con(L).
```

where the delay condition `emptylist(Y)` is enabled when the constraint $Y = []$ is implied by the current constraint store and the delay condition `nonemptylist(Y)` is enabled when Y is constrained to be a non-empty list by the current constraint store.

Both these delay conditions are a type of *ask* delay condition. An ask delay condition has the form $ask(C)$ and is enabled when the constraint store implies the constraint C. For instance, `emptylist(Y)` is equivalent to $ask(Y = [])$. Ask conditions may also involve local variables. These are indicated by existentially quantified ($\exists$) variables in the constraint C. The delay condition `nonemptylist(Y)` is equivalent to $ask(\exists E\ \exists L\ Y = [E|L])$, since this is enabled whenever there exists values for E and L such that the constraint store implies $Y = [E|L]$.

Apart from the ask-based delay conditions, another delay condition widely used in CCP languages is the *tell* condition. This is written $tell(C)$ and is enabled if constraint C is consistent with the global store. If the guard is enabled, another effect of $tell(C)$ is to add C to the constraint store. A typical guard consists of an ask and tell condition.

A major difference between CCP and CLP languages is how multiple rules defining the same predicate are handled. CLP languages, employ don't know non-determinism, trying each rule until an answer is found. In CCP languages, on the other hand, the evaluation mechanism delays choosing which rule to use until at least one of the guards is enabled. If more than one rule has its guard enabled, the evaluation mechanism arbitrarily chooses one of those rules to rewrite the literal with. Regardless of what happens in the future, it never goes back to try the other rules whose guard is enabled or those rules whose guard is still not enabled. It is the programmer's responsibility to provide each rule with a guard that ensures that once it is enabled, then this is the correct rule to choose. Thus CCP languages provide *don't care non-determinism*—if more than one guard is enabled we don't care which of the corresponding rules are used since any will be correct.

Example 12.2

The following program to merge two lists into a third list is a good example of the use of don't care non-determinism. The first rule is enabled whenever there is an element in the first list, and places this element on the merged list. The second rule is enabled when the first list is empty. In this case, the merged list is set to the second list. The third and fourth rule are analogous, except that they apply to the second list rather than the first.

```
merge(XL,YL,ZL) :- nonemptylist(XL) |
     XL=[X|XL1], ZL=[X|ZL1], merge(XL1,YL,ZL1). (M1)
merge(XL,YL,ZL) :- emptylist(XL) | ZL=YL.       (M2)
merge(XL,YL,ZL) :- nonemptylist(YL) |
     YL=[Y|YL1], ZL=[Y|ZL1], merge(XL,YL1,ZL1). (M3)
merge(XL,YL,ZL) :- emptylist(YL) | ZL=XL.       (M4)
```

If more than one rule is enabled, then we do not mind which one is chosen. For example, if the constraint store contains

$$XL = [a] \wedge YL = [b]$$

then the user-defined constraint merge(XL,YL,ZL) may be rewritten with either the first or third rule. Two possible successful simplified derivations are:

$$\begin{gathered}
\langle merge(XL, YL, ZL) \mid XL = [a] \wedge YL = [b]\rangle \\
\Downarrow\ M1 \\
\langle merge([\,], YL, ZL1) \mid XL = [a] \wedge YL = [b] \wedge ZL = [a|ZL1]\rangle \\
\Downarrow\ M2 \\
\langle \square \mid XL = [a] \wedge YL = [b] \wedge ZL = [a, b]\rangle
\end{gathered}$$

and

$$\begin{gathered}
\langle merge(XL, YL, ZL) \mid XL = [a] \wedge YL = [b]\rangle \\
\Downarrow\ M3 \\
\langle merge(XL, [\,], ZL1) \mid XL = [a] \wedge YL = [b] \wedge ZL = [b|ZL1]\rangle \\
\Downarrow\ M1 \\
\langle merge([\,], [\,], ZL2) \mid XL = [a] \wedge YL = [b] \wedge ZL = [b, a|ZL2]\rangle \\
\Downarrow\ M4 \\
\langle \square \mid XL = [a] \wedge YL = [b] \wedge ZL = [b, a]\rangle
\end{gathered}$$

The two answers are different, but equivalent if we *don't care* about the order of elements occurring in ZL.

The CCP paradigm is not well suited for solving combinatorial search and optimization problems, since it does not directly provide don't know non-determinism. Some recent constraint programming languages, such as Oz, provide both don't know and don't care non-determinism as well as guards, effectively combining the CLP and CCP paradigms. We shall discuss Oz further in Section 12.4.

We also note that, as long as the delay conditions and run-time tests are sufficiently expressive, it is possible to translate CCP programs into CLP programs which use delay conditions to mimic the guards and use the cut to implement don't care non-determinism.

The CCP paradigm arose from the concurrent logic programming paradigm whose major application area was the implementation of "reactive" programs and systems. Conventional programs are designed to take an input and to compute some output. On the other hand a program which behaves reactively is not required to compute a final result but rather interacts with its environment. Reactive programs are employed in operating systems, database and transaction handling systems, for example.

The CCP paradigm is thus designed for building reactive programs. In particular, this means it is suitable for building incremental constraint solvers since these are a good example of a reactive system. After all, the role of an incremental constraint solver is simply to react to constraints sent by the CLP system, its "environment."

12.3 Constraint Handling Rules

Constraint handling rules (CHR) are closely related to the CCP paradigm. They are specifically designed for writing constraint solvers and consist of multi-headed guarded rules which repeatedly rewrite the constraint store until it is in a solved form, that is to say, when further rewriting does not change the store.

For example, the following CHRs define a partial order constraint $\leq$.

```
reflexivity @  X ≤ Y         <=> X = Y | true.
antisymmetry @ X ≤ Y, Y ≤ X <=> true | X = Y.
transitivity @ X ≤ Y, Y ≤ Z ==> true | X ≤ Z.
```

The first rule (named `reflexivity`) states that if the store implies that $X = Y$, then $X \leq Y$ can be simplified to true. The second rule enforces antisymmetry, that is, if both $X \leq Y$ and $Y \leq X$ appear in the store, they can be replaced by $X = Y$. The third rule implements transitivity, if both $X \leq Y$ and $Y \leq Z$ appear in the store then we add $X \leq Z$. The first two rules are used to simplify the constraint store, while the third rule adds the logical consequences of constraints in the store. This is similar to propagation.

Constraint handling rules manipulate a constraint store which is treated as a set of primitive constraints. The evaluation mechanism finds primitive constraints in the store which match the pattern on the left hand side of the rule and whose guard (the constraint to the left of the "|") is implied by the constraint store.

$$\begin{array}{c}
\{\underline{A \leq B, B \leq C}, C \leq A\} \\
\downarrow \textit{ transitivity} \\
\{A \leq B, B \leq C, \underline{C \leq A, A \leq C}\} \\
\downarrow \textit{ antisymmetry} \\
\{\underline{A \leq B, B \leq C}, C = A\} \\
\downarrow \textit{ antisymmetry} \\
\{A = B, C = A\}
\end{array}$$

Figure 12.1 CHR evaluation of partial order constraints.

Rewriting takes into account equality constraints for the purposes of matching and implication. For example, it can determine that $A \leq B, B \leq C$ matches the pattern for antisymmetry when $C = A$ is present. A simplification rule (<=>) replaces the matching primitive constraints with the constraint on the right hand side. A propagation rule adds the constraint on the right hand side to the constraint store.

The constraint store $\{A \leq B, B \leq C, C \leq A\}$ is rewritten as illustrated in Figure 12.1.

Constraint handling rules, when combined with a programming language, provide a very straightforward way of defining constraint solvers. Solvers based on a normal form can be implemented straightforwardly using CHRs to place constraints in normal form. Incomplete constraint solvers for complex forms of constraints can be implemented by just giving cases where the solver writer can determine that information can be propagated. CHRs have been used to implement a number of solvers for a wide variety of constraint domains. The ease of writing allows the solver to be highly specialized for the form of constraints expected in an application.

The programmer can use CHRs within a CLP system to provide sophisticated constraint solving behaviour. For example, in Section 8.5.1 we added redundant constraints to the program for scheduling in order to reduce the search space. We know that if tasks with start times $TS_1, \ldots, TS_n$ all require the same machine and have durations $D_1, \ldots, D_n$ and each must finish before the task with start time TS can start, then we can add the constraint

$$TS \geq D_1 + \cdots + D_n + \text{minimum}\{TS_1, \ldots, TS_n\}.$$

CHRs are ideal for implementing these redundant constraints and keeping them updated whenever we make choices about the relative order of tasks on the same machine.

We represent precedence relations using a CHR constraint `pred` as follows. `pred(TS`$_1$`, D`$_1$`, M`$_1$`, TS`$_2$`)` represents that task t_1, with start time TS_1, duration D_1 and machine M_1, must be completed before task t_2 with start time TS_2. The redundant constraints are generated in the form `machpred(MinT, ToTD, M, TS)` indicating that there are tasks with minimum start time $MinT$ and total duration $TotD$ all requiring machine M which must be completed before a task with start time TS.

By replacing the inequalities generated in prectask and exclude by precedence constraints of the form pred(TS$_1$, D$_1$, M$_1$, TS$_2$) the following CHRs can be used to add the precedence and redundant constraints.

```
transitivity @ pred(TS1,D1,M1,TS2), pred(TS2,_,_,TS3) ==> true |
     pred(TS1, D1, M1, TS3).
machine @ pred(TS1,D1,M1,TS2) ==> true | machpred(TS1,D1,M1,TS2).

constraint @ machpred(MinTS,TotD,_,TS) ==> true | TS ≥ MinTS + TotD.
join @ machpred(Min1,Tot1,M,TS), machpred(Min2,Tot2,M,TS) <=> true |
     Min3 = minimum(Min1,Min2),
     Tot3 = Tot1 + Tot2,
     machpred(Min3,Tot3,M,TS).
```

The first CHR transitively builds the precedence relation between tasks. The second CHR creates a machpred constraint for each precedence constraint. The third CHR adds the constraint on the times, namely that the start time TS must be greater than the minimum of the start times plus the total duration. Note that it also has the effect of adding the constraints from each precedence relation because of the rewriting

$$pred(TS1, D1, M1, TS2) \rightarrow machpred(TS1, D1, M1, TS2) \rightarrow TS2 \geq TS1 + D1.$$

The final CHR combines two machpred constraints for the same machine and final task, and creates a new one which sums the durations and takes the minimum of the start times. It replaces two machpred literals by one.

For the example problem of Section 8.5, the precedences user-defined constraints creates the following pred literals:

$$\{pred(TS_1, 3, m1, TS_5), pred(TS_1, 3, m1, TS_6),$$
$$pred(TS_4, 6, m2, TS_3), pred(TS_5, 3, m2, TS_3)\}$$

The first CHR uses the first and last literal to create $pred(TS_1, 3, m1, TS_3)$ because, from transitivity, task $j1$ must be before task $j3$ (through $j5$). The second CHR creates machpred literals for each of these pred literals. The fourth CHR performs one simplification, the literals $machpred(TS_4, 6, m2, TS_3)$ and $machpred(TS_5, 3, m2, TS_3)$ are replaced by $machpred(T, 9, m2, TS_3)$ where T is constrained to be the minimum of TS_5 and TS_4. The third CHR creates inequalities over the times corresponding to the machpred literals:

$$TS_1 + 3 \leq TS_5 \land TS_1 + 3 \leq TS_6 \land TS_4 + 6 \leq TS_3 \land TS_5 + 3 \leq TS_3 \land$$
$$T = minimum(TS_5, TS_4) \land T + 9 \leq TS_3.$$

We assume that the third rule is always used before the fourth rule replaces a machpred literal. The propagation solver obtains the range domain

$$D(TS_1) = [0..6], \quad D(TS_2) = [0..12], \quad D(TS_3) = [9..12],$$
$$D(TS_4) = [0..6], \quad D(TS_5) = [3..9], \quad D(TS_6) = [3..16].$$

The information about TS_3 is better than that obtained without adding the redundant information.

The machine exclusion constraints add further pred literals which cause new inequalities on start times to be added and the machpred literals to be further collected. For this small example, this causes no further difference in behaviour.

This approach using CHRs automatically maintains the predecessors of a task, given the current choice of ordering of tasks on the same machine, and adds the redundant information about predecessors on the same machine after each choice is made. It illustrates how CHRs, when used in conjunction with CLP, can be a very powerful tool for the constraint programmer. With little programming effort powerful redundant constraints can be added to the model which are maintained throughout the evaluation.

12.4 Functional Languages

We now look at how constraint programming has been integrated into a number of other paradigms. Like logic programming languages, functional languages are declarative—that is to say, the programmer specifies what to compute, not how to compute it. Given that constraints are declarative, and that the marriage of constraints and logic programming has been so successful, it is natural to ask whether constraints and functional languages can be combined. This has been an area of active research, and there are currently three main approaches.

The first, and one of the simplest approaches, is to embed primitive constraints and user-defined constraints into a functional language by viewing them as a function which returns a list or set of answers. As the set may be infinite, a lazy functional language is needed. The problem with this approach is that it loses much of the appeal of the constraint programming paradigm, since the programmer must explicitly program constraint solving and search using functions. Furthermore, there are difficulties dealing with variables in the answers.

The dual approach in which functions are embedded into constraint logic programming has also been tried. We note that CLP languages do provide functions—namely functions such as "+" in the underlying constraint domain. However, for many problems it is natural to want user-defined functions, such as a function *append* to concatenate two lists. At first this seems easy enough, after all, an n-ary function can always be thought of as a relation with an extra argument representing the result of the function. Ideally, the user should be able to define at least first order functions using syntactic sugar. For example, the functional definition of how to concatenate two lists,

```
append([],L)          =   L
append([X|LX], L)     =   [X| append(LX,L)]
```

can be straightforwardly translated into the standard append relation,

```
append([],L, Ans)         :-   Ans = L
append([X|LX], L, Ans)    :-   Ans = [X|Ans1], append(LX,L,Ans1).
```

Unfortunately there are difficulties with this approach. First, one would like constraint solving over user-defined functions to be as powerful as that over the pre-defined functions. For instance, you might write the constraint

$$X = [1], append(Y, X) = Y.$$

Unfortunately no straightforward evaluation mechanism is going to detect failure. The problem is that the function definition is operationally efficient for one mode of usage but probably not for all. A related problem is that of non-determinism—what should the evaluation mechanism do to answer the constraint

$$append(X, Y) = [1, 2, 3, 4, 5]?$$

One approach has been to extend the evaluation mechanism by providing term-rewriting for user-defined functions. Another approach has been to delay function evaluation until enough input arguments are fixed.

There is yet a third approach to combining constraints and functions. The third approach is designed to be an equal marriage of constraints and functions, unlike the previous two approaches in which constraints are built on top of a functional language, or functions are built on top of a CLP language. The key idea is to extend the lambda calculus, the rewrite system providing the evaluation mechanism for functional languages, by including a global constraint store. Function application may now add constraints to the store. The chief difficulty is determining how the the contents of the constraint store can be used to control program evaluation.

One method, known as *constrained lambda calculus*, uses the store to determine the values of variables. If the store determines a variable has a fixed value then it replaces the variable by its value throughout the lambda expression. The problem with this approach is that the constraint store is rather passive—it cannot be actively queried to guide the search.

Another approach is based on the CCP paradigm. The idea is that lambda expressions may have a guard associated with them. They may only be evaluated if the constraint store enables the guard. Variables may range over lambda expressions, and in particular procedure definitions. Thus, in this approach, as well as a constraint store there is a *procedure store* which holds the definition of procedures and their binding to "procedure variables". Not surprisingly, evaluation of a procedure is delayed until its definition is known.

This approach was pioneered in the language Oz. Oz includes constraint solvers for tree constraints and finite domain constraints, and supports the addition of new constraint solvers through its object oriented nature. An example of an Oz program for the send-more-money problem described in Section 8.1 is given below:

```
proc {SendMoreMoney Sol}
   [S E N D M O R Y] = !Sol
   in
   Sol ::: 0#9
   {FD.alldifferent Sol}

   M \=: 0  S \=: 0
   1000*S + 100*E + 10*N + D +
   1000*M + 100*D + 10*R + E
   =: 10000*M + 1000*O + 100*N + 10*E + Y
   {Forall Sol FD.enum}
end
```

The definition in effect defines the procedure variable *SendMoreMoney* which takes a single argument *Sol*. *Sol* is made up of a list $[S, E, N, D, M, O, R, Y]$ of local variables which are all digits, and constrained to be all different using the `alldifferent` method of the module *FD* (for finite domains). The arithmetic constraints are straightforward. The final statement is analogous to a `labelling` step. It adds an enumerator *FD.enum* to each variable. Notice the similarity in form to the CLP program given in Section 8.1.

Search in Oz is implemented quite differently from that in CLP languages. Search is programmable rather than being by default left-to-right depth-first. A search procedure must be used to explore the search tree. This is achieved by setting the query of the *Search* object and asking it for a next solution. For example,

```
{Search query(SendMoreMoney)}
{Search next}
```

The *Search* object encodes a search strategy. By elaborating the computation of *SendMoreMoney*, it finds a suspended choice corresponding to the enumeration routine. The search procedure can then choose how to explore this suspended choice. This approach relies on the ability to create named local computation spaces and pass them as arguments to procedures.

Oz generalizes the CLP and CCP paradigms and provides a very flexible approach to constraint programming by allowing the use of arbitrary search strategies. Unfortunately, since the search strategy is arbitrary, the techniques discussed in Chapter 10 for efficient incremental backtracking with constraint solving are not always applicable. Hence, the price of this flexibility may be a significant performance overhead if the constraint store needs to be copied during search.

12.5 Term Rewriting

Constraint programming has also been integrated into another declarative paradigm, term rewriting. Term rewriting uses pattern matching to apply *rewrite rules* to simplify terms. For example the following rewrite rules

$$\begin{array}{rcl}
\neg\neg X & \rightarrow & X \\
\neg(X \vee Y) & \rightarrow & (\neg X) \wedge (\neg Y) \\
\neg(X \wedge Y) & \rightarrow & (\neg X) \vee (\neg Y) \\
Z \vee (X \wedge Y) & \rightarrow & (Z \vee X) \wedge (Z \vee Y) \\
(X \wedge Y) \vee Z & \rightarrow & (X \vee Z) \wedge (Y \vee Z)
\end{array}$$

can be used to rewrite a Boolean formula into its conjunctive normal form. For instance, consider the Boolean formula $\neg(X \vee (Y \wedge \neg Z))$. This can be rewritten as follows where the subexpression which is rewritten is underlined.

$$\begin{array}{c}
\underline{\neg(X \vee (Y \wedge \neg Z))} \\
\downarrow \\
(\neg X) \wedge \underline{\neg(Y \wedge \neg Z)} \\
\downarrow \\
(\neg X) \wedge ((\neg Y) \vee \underline{\neg(\neg Z))} \\
\downarrow \\
(\neg X) \wedge ((\neg Y) \vee Z)
\end{array}$$

Note that in term rewriting, rewrite rules may be applied in any order and at any position they match in the term. A major concern is, therefore, that whichever rules are chosen and in whatever order they are used, the same result is obtained.

Term rewriting have been widely used to encode decision procedures for determining if equations hold under certain axioms. The idea is that both sides of the equation are rewritten using the rewrite rules, and they are equal if they are both rewritten to the same term. As an example, the above rules provide a partial axiomatization of Boolean algebras. They can determine that two Boolean expressions are equal whenever they can be rewritten into the same conjunctive normal form. Term rewriting has also been used for specification of abstract data types and as the basis for programming languages.

There have been two main approaches to combining term rewriting and constraints. The first approach, called *augmented term rewriting*, has many similarities to CLP. Like CLP, the programmer provides user-defined constraints, specified by rewrite rules, and execution proceeds by rewriting these rules. However, unlike CLP languages, there is no underlying constraint solver—the constraint solver is written in the same formalism.

Augmented term rewriting extends standard term rewriting in two main ways. The first extension is to allow non-local substitution. This is provided by the special expression X *is* e which is processed by being rewritten to *true* and by replacing X by e throughout the term being rewritten. For instance, the expression,

$$X \text{ is } Y + 5 \wedge X = 2 \times Y$$

can be rewritten to

$$true \wedge Y + 5 = 2 \times Y.$$

Substitution is provided because it allows the easy definition of solvers, such as Gauss-Jordan, which are based on variable elimination. The second extension is to allow new local variables to be introduced in the right hand side of the rewrite rules. This is important because it allows user-defined constraints to be specified in a natural manner.

The following example, taken from [86], illustrates augmented term rewriting.[1] The program specifies how to append two lists. Note that append is used as an infix operator.

```
[] append L       →  L
[X|L1] append L2  →  [X|(L1 append L2)].
```

The programmer must also specify how constraint solving works for lists. The following rewrite rules define list equality:

```
[] = []           →  true
[X|L1] = [Y|L2]   →  X=Y  ∧  L1=L2.
```

The equality relationship for the integers is built in, but is conceptually defined by

```
1 = 1  →  true
2 = 2  →  true
3 = 3  →  true
       ⋮
```

The conjunction operator "∧" has the following rewrite rule

```
true ∧ X  →  X.
```

Now consider the expression $([1,2] \; append \; x) = [1,2,3,4,5]$. This will be rewritten as illustrated in Figure 12.2 We can also solve for x in the expression $([1,2] \; append \; [3,4,5]) = x$.

However, the expression $(x \; append \; y) = [1,2,3,4,5]$ cannot be rewritten. There is no mechanism to solve queries of this form as the operational semantics does

1. The syntax is slightly modified to fit that of rewrite systems

$$\begin{array}{c}
\underline{([1,2]\ \mathit{append}\ x)} = [1,2,3,4,5] \\
\downarrow \\
\underline{[1|([2]\ \mathit{append}\ x)] = [1,2,3,4,5]} \\
\downarrow \\
\underline{(1 = 1)}\ \wedge\ ([2]\ \mathit{append}\ x) = [2,3,4,5] \\
\downarrow \\
\underline{\mathit{true}\ \wedge\ ([2]\ \mathit{append}\ x) = [2,3,4,5]} \\
\downarrow \\
\underline{([2]\ \mathit{append}\ x)} = [2,3,4,5] \\
\downarrow \\
\underline{[2|([\,]\ \mathit{append}\ x)] = [2,3,4,5]} \\
\downarrow \\
\underline{(2 = 2)}\ \wedge\ ([\,]\ \mathit{append}\ x) = [3,4,5] \\
\downarrow \\
\underline{\mathit{true}\ \wedge\ ([\,]\ \mathit{append}\ x) = [3,4,5]} \\
\downarrow \\
\underline{([\,]\ \mathit{append}\ x)} = [3,4,5] \\
\downarrow \\
x = [3,4,5]
\end{array}$$

Figure 12.2 Rewriting $([1,2]\ \mathit{append}\ x) = [1,2,3,4,5]$.

not include search since the operational mechanism employs "don't care non-determinism." This is one of the main drawbacks of augmented term rewriting and means that it is not suitable for solving combinatorial problems.

On the other hand, a nice feature of augmented term rewriting is that term rewriting provides a uniform mechanism for both constraint solving and evaluation.

The other approach to combining constraints with term rewriting is *constrained term rewriting.* The idea here is to add constraints to rewrite rules which specify when they may be used for rewriting. As a simple example, consider the rewrite rule for commutativity of "+,"

$$X + Y \rightarrow Y + X.$$

Unfortunately, such a rule leads to non-termination since it can be applied an infinite number of times. However, if we use the constrained rewrite rule

$$X + Y \rightarrow Y + X \mid\mid X \succ Y$$

then we no longer have this problem. We have added a condition $X \succ Y$ that ensures we can only apply this rule when X is greater than Y where $\succ$ is some fixed total ordering on the expressions. At each step in rewriting some underlying constraint solver is used to verify that the constraint associated with the rule is satisfied by the expressions mapped to the variables in the rule.

Constrained term rewriting is primarily intended for implementing automatic theorem provers. Apart from the application area, the main differences to CLP are that: (1) there is no global constraint store, (2) matching is used when applying rules rather than equating arguments, (3) it rewrites expressions, rather than goals, and (4) rules are applied using a "don't care non-determinism." Because of (4) and because (2) can be implemented by using delay condition on rule application, constrained term rewriting is more closely related to CCP languages than to CLP languages.

12.6 Imperative Programming Languages

One interesting area of research is to integrate constraints into traditional imperative programming languages, in particular object oriented languages. The motivation for doing so is clear. Constraints allow the programmer to declaratively specify what they want, allowing simple concise programs. This is particularly useful in incremental contexts where repeated changes to one variable must be communicated to other variables. Conversely, the imperative paradigm allows the programmer to write code which is guaranteed to behave efficiently and which can be easily integrated into existing applications. However, the integration of the two paradigms is not easy. In contrast to the constraint paradigm, the imperative paradigm allows a variable to have different values at different times in the program execution. It also requires that a variable has a fixed value when it is used.

An interesting approach is used in the *Kaleidoscope* family of languages which generalize both the constraint and imperative paradigms. In this paradigm there is a single global constraint store. As in the imperative paradigm, at different times during the execution of a program, the same variable may appear to have different values. Of course, giving a variable more than one value is anathema to the constraint paradigm. However, the Kaleidoscope family of languages circumvent this problem by regarding a program variable as a sequence of logical variables. At different points in time, the program variable stands for different logical variables, essentially subscripted by the time of execution. Thus the assignment

```
X := X+1;
```

is really understood as representing the constraint

$$X_{t+1} = X_t + 1.$$

In the Kaleidoscope paradigm, every variable has a fixed value at all points in the execution. This is achieved by using a hierarchy of constraints in which, each constraint has an associated weight. Constraints weighted with `always` must be satisfied at all time points, this is the strongest requirement. While for constraints with other weights, those with heavier weights are satisfied in preference to those with lighter weights. There is a weak default constraint which ensures that variables remain the same over time whenever reasonable, that is $X_{t+1} = X_t$.

As a simple example, consider the following Kaleidoscope'90 program:

```
always: X = Y;
X := 10;
while mouse.down do
  X := mouse.posn.x;
end;
print Y;
```

The program specifies a global constraint $X = Y$ and then assigns values to X. This will automatically update the value of Y so as to maintain the constraint.

The Kaleidescope approach to the integration of constraints and the imperative programming language relies much on the CLP paradigm. Like CLP there is a global constraint store and incremental constraint solving algorithms must be employed. However, the Kaleidescope approach deliberately eschews search and backtracking, in order to remain deterministic. This makes it suitable for incremental applications such as graphical user interface building, but not suitable for solving combinatorial search or optimization problems.

12.7 Constraint Solving Toolkits

Constraint solving toolkits provide another approach for integrating constraints into a traditional programming language. Although not an equal marriage between constraints and imperative programming like the Kaleidescope approach, the integration provided by these toolkits is quite natural. This is possible because of the object oriented nature of the implementation languages which allow constraints, (logical) variables and solvers to be first class entities. However, the toolkits do not provide a single unified paradigm like constraint logic programming. Rather the integration embeds constraint programming into a host imperative programming language in a simple way.

These toolkits are an offshoot of the CLP paradigm and employ incremental constraint solving algorithms similar to those used in CLP systems. Many of the solvers also employ the same model of computation as CLP in which there is a single global store to which constraints are added and non-deterministic search is used to find solutions to a given goal.

Current toolkits tend to be problem domain focused, for instance on graphical applications or scheduling applications. We shall look at two example toolkits, in order to give some feel for their power and use.

Our first example is the ILOG SOLVER toolkit which is designed for solving finite domain problems. The following C++ program from [107] employs the ILOG SOLVER toolkit to solve the N-queens problem (described in Example 3.2).

```
#include <ilsolver/ilcint.h>

void main() {
  IlcInit();
  int i, nqueen = 1000;
  IlcIntVarArray x(nqueen,0,nqueen-1),
                 x1(nqueen), x2(nqueen);
  for (i=0; i<nqueen; i++) {
    x1[i] = x[i] + i;
    x2[i] = x[i] - i;
  }
  IlcTell(IlcAllDiff(x));
  IlcTell(IlcAllDiff(x1));
  IlcTell(IlcAllDiff(x2));
  IlcSolve(IlcGenerate(x, IlcChooseMinSize));
  cout << x << endl; //prints
  IlcEnd();
}
```

The initial call to `IlcInit()` initializes the internal data used by the constraint solver and evaluation mechanism while the call to `IlcEnd` releases the memory used. The call to `IlcIntVarArray` declares `x`, `x1` and `x2` to be arrays of size *nqueen* of constrained integer variables or expressions. The additional arguments for the declaration of `x` specify the domain of the variables to be $0, \ldots, nqueen-1$. The **for** loop constructs expressions which are placed in the arrays `x1` and `x2`. Note the use of overloading of the standard arithmetic operators. The call to `IlcAllDiff` creates a constraint which specifies that all expressions in the input array are different, that is no two expressions are equal. The `IlcTell` function adds a constraint to the global store. Finally, the function `IlcSolve` searches for one solution to a goal while `IlcGenerate` is a goal which is evaluated using standard labelling techniques. The argument `IlcChooseMinSize` is the name of a strategy to be employed by `IlcGenerate` when choosing which variable to label. In this case, the strategy is to choose the variable with smallest domain.

This program is rather typical of the way in which the ILOG SOLVER toolkit is used. First, the constraints are built using overloading and the control facilities of C++, then they are placed in the store, and finally non-deterministic search is used to find the a solution or minimum to the constraints in the store.

This indicates one difference between the toolkit approach and CLP. In the CLP paradigm user-defined constraints are used to collect the primitive constraints which are added to the constraint store. In a toolkit this is done by using the control and abstraction facilities of the host language. A consequence of this is that it is difficult to dynamically control the addition of the constraints in the same way that you can in a CLP system. Another, consequence is that debugging and optimization is at the level of the host language rather than at the level of the constraints being used to model the problem.

On the other hand, the toolkit approach has at least two advantages over earlier CLP systems. First, some toolkits allow the programmer to define new primitive constraints by extending the solver and search mechanism to handle these constraints. This provides considerable flexibility and potential efficiency gains for the programmer. Second, code developed using a toolkit can be more easily integrated into a large application.

However, it is worth reemphasizing that these object oriented toolkits are based on the CLP paradigm and most of the modelling and programming techniques we have described for CLP can be employed when using a toolkit. These toolkits are therefore very different in nature from the traditional mathematical software libraries of numerical algorithms provided in such languages as C or FORTRAN. In mathematical software libraries, constraints, variables and search strategies are not explicit in the program but rather implicitly represented in the implementation by, for example, an array of coefficients. Thus mathematical software libraries require the programmer to work at a very different level. Of course, it is possible to take an existing mathematical software library and encapsulate it using a high level interface.

The second major application area of constraint solving toolkits is for graphical applications. Typical graphical applications have three important requirements which are not met in the usual CLP paradigm. First, they require arbitrary removal of constraints since the user can select any graphic object and delete it, thus requiring its associated constraints to be removed. Second, rather than just testing for satisfiability, graphical applications require a solution to be found to the current constraint store, as this solution determines how the graphics will be displayed on the screen. However, not just any solution will do, since it is unsatisfactory if objects unnecessarily move all over the screen whenever a constraint is added. Various techniques, including finding the solution as close as possible to the old solution and the use of hierarchical constraint systems in which some constraints are optional (similar to that described for Kaleidoscope), have been used. Third, the toolkits must support very fast direct manipulation. Typically this is done by looking at the current constraints, determining dependencies between them and then "compiling" a function which can quickly compute the changes to the last solution if the values for the fixed (small) set of variables being manipulated are modified.

QOCA is an example of C++ toolkit for graphical applications. The following C++ program makes use of QOCA to perform the same task as the example Kaleidoscope'90 program from the previous section.

```
CFloatHandle x,y;
LinEqSolver solver;
LinConstraint eqxy = (1.0*x - 1.0*y == 0.0);
```

```
solver.AddConstraint(eqxy);
x.SetValue(10.0);
solver.AddEditVar(x);
while (mouse.down) {
   x.SetDesValue(mouse.posn.x);
   solver.Resolve();
   }
solver.ClearVarsOfInterest();
cout << y.GetValue() << endl;
```

The call to `solver.AddConstraint` adds the linear constraint `eqxy` which constrains the values of the variables `x` and `y`. The call `solver.AddEditVar` indicates that variable `x` will be the subject of direct manipulation. The first call to `solver.Resolve` generates data structures which are used in subsequent calls to `solver.Resolve` to solve the constraints for different values of `x`.

Toolkits for graphical applications are more closely related to the Kaleidescope paradigm than to the CLP paradigm. In particular, they do not provide search and variables always have a fixed value which may change over time.

12.8 Mathematical Languages

Finally, we shall look at mathematical languages. We use this term to cover two types of language for specifying and dealing with constraints: *mathematical modelling languages* and *symbolic algebra packages*. Mathematical modelling languages provide a way of specifying large conjunctions of primitive constraints, in CLP terms a user-defined constraint, in a way that is natural and easy for the modeller to express, and which can be automatically translated into a form that can be handled by an underlying constraint solver. Symbolic algebra packages support symbolic manipulation of mathematical expressions, such as constraint solving, integration, differentiation, and usually provide numerical manipulations, such as numeric constraint solving, and graphical display of functions.

12.8.1 Modelling Languages

The process of modelling can be thought of as providing an abstract system of primitive constraints that represent the general form of the problem to be solved. Questions are then asked by specifying data for a particular problem instance. The model and data are converted to a solver digestible form and fed to the solver to find the answer.

There are a number of popular modelling languages used in the operations research and mathematics community. One example language is AMPL, a language for specifying linear arithmetic optimization problems in a convenient human readable form.

One of the most pleasing features of mathematical modelling languages are their broad facilities for handling sets and variables that are indexed by sets, that is arrays. This is a core aspect of these languages, designed to make it simple to specify large numbers of linear constraints succinctly. The languages themselves are usually designed so that constraints are specified in a syntax which is close to mathematical notation.

Another feature of mathematical modelling language is that data for different problem instances can be input separately from the model and different optimization functions can be specified for the same constraints, so a single model can be used to specify many different problems.

As an example, consider the following AMPL model of the problem discussed in Example 6.4 of determining temperatures in a finite element description of a metal plate described using $W \times H$ elements, in which the top edge is set to one temperature and the remaining edges to another.

```
param W > 2; # width of plate
param H > 2; # height of plate

param toptemp; # temperature at top of plate
param edgetemp; # temperature on other edges

var Temp {1..W, 1..H} # temperature at positions of the plate

subject to topedge {c in 1..W}: Temp[1,c] = toptemp;
subject to leftedge {r in 2..H}: Temp[r,1] = edgetemp;
subject to rightedge {r in 2..H}: Temp[r,W] = edgetemp;
subject to bottomedge {c in 1..W}: Temp[H,c] = edgetemp;

subject to dirichlet {r in 2..H-1, c in 2..W-1}:
4*Temp[r,c] = Temp[r-1,c] + Temp[r,c-1] +
      Temp[r,c+1] + Temp[r+1,c];

minimize totaltemp: sum {c in 1..W, r in 1..H} Temp[c,r];
```

Although the original problem is a satisfiability problem we are forced to add a dummy minimization goal so that it can be solved by AMPL. The 6×7 instance of the problem illustrated in Figure 6.1 can be specified by the data file

```
param W = 7;
param H = 6;
param toptemp = 100;
param edgetemp = 0;
```

Mathematical modelling languages are excellent for solving the problems that fit within the class they are designed to model. One advantage of modelling languages is that they can usually be used with a variety of linear arithmetic optimization packages, for example MINOS and CPlex. As we have discussed, another advantage is that they neatly differentiate the model of a problem from the constraints that describe the data for a particular instance of the problem.

The restrictions of mathematical modelling languages arise from the underlying solver capabilities. These languages can usually only be used to specify a conjunction of linear constraints together with a linear optimization function. Although these constraints can be built in a number of flexible and intuitive ways the lack of nonlinear constraint handling means that, for example, the `mortgage` program cannot be defined.

While mathematical modelling languages provide excellent facilities for handling sets and arrays, they are more difficult to use with other types of data. For example, even though all the arithmetic constraints produced in the tree layout program of Section 6.6 are linear, and so it is a linear optimization problem, it is difficult to see how to use a mathematical modelling language such as AMPL to construct these constraints from the initial tree description, since it is not a general purpose programming language.

Because mathematical modelling languages merely specify a conjunction of constraints and an optimization function, and then send them to a remote solver there is no way to specify search strategies within them. Choices need to be converted to a conjunctive form using methods analogous to those discussed in Section 8.7 and Exercise P8.16. If the underlying constraint solver uses search to solve such constraints then it is largely beyond the control of the modeller.

We foresee CLP languages becoming the mathematical modelling languages of the future because of their greater flexibility (they are full programming languages) and their ability to specify search within the same language. However for this to happen, CLP languages will need to incorporate the excellent handling of sets and arrays of current mathematical modelling languages and provide the ability to interface with other solvers.

12.8.2 Symbolic Algebra Packages

As their name suggests, symbolic algebra packages are designed to perform symbolic algebra manipulation automatically. Their main use is as a toolkit for mathematicians and scientists who are interested in solving algebraic problems. Many of the facilities in a symbolic algebra package are constraint operations. Algebraic expressions can be constructed, rewritten and simplified. Arithmetic constraints can be specified, and solved by the package. Such packages usually include sophisticated handling of nonlinear constraints.

Mathematica is an example of a symbolic algebra package which also includes a completely general programming language. It allows programming using functional, procedural and term rewriting paradigms. An example of a Mathematica program to solve problems about mortgages (when the number of time periods is fixed) is:

```
mortgage[p_, t_Integer?Positive, i_, r_] :=
    mortgage[p * (1 + i) - r, (t-1), i, r];
mortgage[p_, 0, i_, r_] := p;
```

The program defines a function `mortgage[p, t, i, r]` which returns the balance at the end of a mortgage for principle p for length t periods with interest rate i and repayment r. It works by constructing a mathematical expression of the balance in terms of the inputs. The expression can then be used to build a constraint which can be passed to the solver. For example the command

```
Solve[mortgage[p, 10, 10/1200, r] == b]
```

asks for the relationship between principal, repayment and balance for a 10 month mortgage at 10% per annum. The answer is

```
          6727499949325600009201 p  - 5357635269256000920l r
{{ b ->   ----------------------      ---------------------   }}.
          6191736422400000000000       51597803520000000000
```

This is the exact rational answer, a floating point representation is determined by

```
N[Solve[mortgage[p, 10, 10/1200, r] == b]]
```

which gives the answer

```
{{ b -> 1.08653 p - 10.3835 r }}.
```

Because Mathematica provides very sophisticated nonlinear constraint solving and simplification, it can be used to answer some difficult questions. For example "What is the interest rate for a mortgage of 3 periods for $1000 with repayment $500 to end up with balance 0?" can be expressed as:

```
N[Solve[mortgage[1000, 3, i, 500] == 0]].
```

Mathematica determines three exact answers (two complex values for i) which are large and complicated expressions. Since we asked for the numeric values we obtain only one

```
{{ i -> 0.233752 }}.
```

Symbolic algebra packages allow very powerful reasoning about arithmetic constraints, particularly nonlinear constraints. They are designed as a support tool for mathematicians. They are not really designed for specifying and solving large systems of constraints. Because symbolic constraint packages like Mathematica include general purpose programming facilities, it is possible to do so, but it is not straightforward. Since there is no implicit constraint store or search mechanism, both must be programmed using the facilities of the language. The constraint solving facilities are not designed for incremental constraint solving and so they can be very slow, especially when solving problems which involve search. Symbolic algebra packages do, however, provide many useful facilities, such as nonlinear constraint solving and simplification, so they are sometimes used within other constraint languages.

12.9 Notes

Clark and Gregory introduced committed choice and don't care non-determinism into Prolog [29]. For more information about concurrent logic languages see the survey by Shapiro [117]. Maher generalized concurrent logic languages to the constraint setting by recognizing that the synchronization operator used in concurrent logic languages can be thought of as constraint entailment [90]. The formal properties of concurrent constraint programming languages [59, 139] have been widely studied since then. In particular, Saraswat has provided elegant theoretical semantics for these languages [115] and was responsible for the term "concurrent constraint programming."

Constraint handling rules were introduced by Frühwirth [51] and are described more fully in [52]. An implementation of constraint handling rules is included in the ECLiPSe system. It also contains many constraint solvers built using constraint handling rules including solvers for Booleans constraints, set constraints, equation solving over real or rational numbers, temporal reasoning and geometric reasoning about rectangles. A number of applications of constraint handling rules are described in [52]. Mark Wallace suggested their use for adding redundant constraints in the scheduling example.

Before the advent of constraint logic programming there was considerable interest in combining the logic and functional programming paradigms—indeed to some extent the generalization of logic programming to allow functions naturally paved the way to the theory of constraint logic programs. See for example the collection of papers in [40]. The first approach to the integration of constraints into functional languages is due to Darlington *et al* and described in [39]. The paper of Crossley *et al* [36] describes the constrained lambda calculus approach in which only definite values can be communicated from the constraint store. The Oz programming language has been developed by Smolka. In addition to incorporating functional and concurrent constraint programming, it also provides objects. It is introduced in [121, 120]. Oz can be obtained via the Web from `http://www.ps.uni-sb.de/oz/`

Augmented term rewriting was introduced by Leler in [86]. Constrained term rewriting generalizes ordered rewriting systems. An introduction is given by Kirchnir [79].

Combining constraint with imperative programming languages has not been investigated very fully. Kaleidoscope [49] developed by Borning and others, is one of the few examples of such languages. The Kaleidoscope approach seems less successful than the constraint toolkit approach in combining the two paradigms, in part because of the conflict between the imperative programmer's desire to fully know the state of every variable during execution and the constraint programming view of complex flow of information. In the toolkit approach the programmer is limited in their handling of constraints and constrained variables by the abstract data type provided by the toolkit.

CHARME [100] and ILOG SOLVER [106] are commercial constraint-solving toolkits for finite domain constraints. They are both offshoots of the CLP language CHIP developed at ECRC and use similar propagation based constraint solving techniques. An object oriented Lisp based toolkit called PECOS [105] was the precursor to ILOG SOLVER.

The use of constraints in computer graphics dates from Sutherland's system SKETCHPAD [133]. Since that time this has been a fertile research area with applications in CAD, simulation, user interfaces and modelling of dynamic physical objects. Two toolkits especially designed for such applications and which employ incremental arithmetic constraint solving techniques developed for CLP are QOCA [64, 16], which is a C++ toolkit providing linear arithmetic constraints, and Ultraviolet [15], which is a Smalltalk toolkit providing local propagation and linear equality solvers.

Mathematical modelling languages were first introduced in the 1970's, replacing matrix generators as the way of specifying large linear programs. The majority of mathematical modelling languages are designed for real linear arithmetic constraints with real linear optimization functions, since these are the constraints supported by the underlying solvers used by the modelling languages. Mixed integer constraints are supported by later modelling languages. AMPL [46] fits in this category and also provides additional forms of modelling such as facilities for modelling graphs. ALICE [83] is a somewhat different mathematical modelling language designed for integer optimization problems. It allows models to be written using abstract functions which may be injective, bijective, etc. Constraint solving is handled by consistency methods such as those discussed in Chapter 3 as well as by reasoning about abstract functions.

The first widely available symbolic algebra packages were MACSYMA [57] and REDUCE [109] developed in the late 1960s. Even these early systems provided a wide range of facilities and were extensible and modifiable. Modern symbolic algebra packages such as Mathematica [143] and MAPLE V [26] enhance earlier packages by including extensive facilities for programming and graphical display of functions.