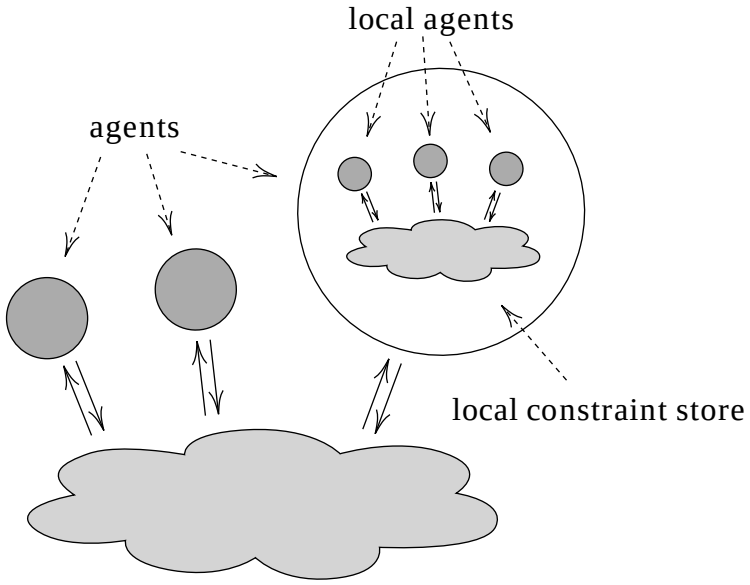

Figure 2.2. An agent with a local constraint store

Let us start with the statement

```
( or(X, Y) | q )
```

The or atom is unfolded, giving

```
( ( X = 1 ? true ; Y = 1 ? true ) | q )
```

Since no other step is possible, we may try the alternatives of the nondeterminate choice in different copies of the closest enclosing clause, which is duplicated as follows.

```
( X = 1 | q
; Y = 1 | q )
```

Other choice statements are handled analogously.

Before leaving the subject of don't know nondeterminism in guards, it should be clarified when alternatives may be tried. (A formal definition is given in Chapter 4.) A local store with associated agents is *stable* if no computation step other than copying in nondeterminate choice is possible (i.e., all agents are inactive), and no such computation step can be made possible by adding constraints to external constraint stores (if any). Alternatives may be tried for the leftmost possible nondeterminate choice in a stable state.

By only executing a nondeterminate choice in a stable state, don't know nondeterministic computations will be synchronised in a concurrent setting in a manner not unlike the synchronisation achieved by conditional or committed choice. For example, the agent

```
member(X, Y)
```

will unfold to

(Y_1 : Y = [X | Y_1] ? true
; X_1, Y_1 : Y = [X_1 | Y_1] ? member(X, Y_1))

By adding constraints to the environment of this agent, it is possible to continue execution without copying, e.g., by adding X = 1 and Y = [2 | W]. Thus, while there are active agents in its environment that may potentially tell constraints on Y, the above agent is unstable.

2.7 BAGOF

Finally, we introduce a statement which builds lists from sequences of alternative results. It provides powerful means of interaction between determinate and nondeterminate code. It is similar to the corresponding construct in Prolog, and a generalisation of the list comprehension primitive found in functional languages (e.g., [Hudak and Wadler 1991]).

A *bagof* statement of the form

bagof(⟨variable⟩, ⟨statement⟩, ⟨variable⟩)

builds a list of the sequence of alternative results from its component statement. The different alternatives for the variable in the first argument will be collected as a list in the variable in the last argument. The statement will be executed within the bagof statement in a manner not unlike the execution of a guard. It is required that the alternatives only restrict the given variable or variables introduced in the component statement. Don't know nondeterminism is not propagated outside it.

For example, the composition

member(X, [a, b, c]), member(X, [b, c, d])

has two alternative results X = b and X = c. By wrapping this composition in a bagof statement, collecting different alternatives for X in Y

bagof(X, (member(X, [a, b, c]), member(X, [b, c, d])), Y)

the result becomes

Y = [b, c]

as could be expected.

Relating this to our previous discussion of stability, the member atoms in

bagof(X, (member(X, Z), member(X, W)), Y)

are unstable (after unfolding) if there are active agents in their environment. If the constraints Z = [a | Z_1] and W = [b | W_1] are added, they become stable, and execution may proceed. After a while, a new unstable state is reached. Then, more information can be added, and so on. Thanks to stability, don't know nondeterministic computations are not affected by the order in which different agents are processed.

Bagof exists in two varieties: ordered (the default) and unordered. The don't know nondeterministic alternatives are, as usual, ordered in the order of clauses in the nondeterminate choice. Thus,

((X = a ; X = b) ; (X = c ; X = d))

generates alternatives for X in the order a, b, c, d. So,

bagof(X, ((X = a ; X = b) ; (X = c ; X = d)), Y)

yields Y = [a, b, c, d]. However,

unordered_bagof(X, ((X = a ; X = b) ; (X = c ; X = d)), Y)

ignores this order, and collects an alternative in the list as soon as it is available. Depending on the implementation, this could lead to a different order, e.g., Y = [d, c, b, a]. See Section 7.3.2 for an (unexpected) application of unordered bagof. Normally, ordered bagof is used. See Section 3.6 for a typical application.

2.8 MORE SYNTACTIC SUGAR

Analogously to what is usually done for functional languages, we now introduce syntactic sugar that is convenient when the guards in choice statements consist mainly of pattern matching against the arguments, as is often the case.

A definition of the form

$p(X_1, \ldots, X_n)$:=
(g_1 % b_1
; …
; g_k % b_k).

where % is either →, |, or ?, may be broken up into several *clauses*

$p(X_1, \ldots, X_n)$:- g_1 % b_1.
…
$p(X_1, \ldots, X_n)$:- g_k % b_k.

which together stand for the above definition. Each clause has a *head*.

The main point of this transformation into *clausal* definitions is that the following additional syntactic sugar may be introduced, which will be exemplified below: (1) Free variables are implicitly hidden, but here the hiding statement encloses the right hand side of the clause (i.e., to the right of ":-"), and not the entire definition. (2) Equality constraints on the arguments in the guard part of a clause may be folded back into the *heads* $p(X_1, \ldots, X_n)$ of these clauses. (3) If the remainder of the guard is "true", it may be omitted. (4) If the guard is "true", and the guard operator is wait "?", the guard operator may be omitted. (5) If the guard operator is omitted, and the body is "true", a clause may be abbreviated to a head.

As an example, the definition

```
member(X, Y) :=
    ( Y1 : Y = [X | Y1] ? true
    ; X1, Y1 : Y = [X1 | Y1] ? member(X, Y1) ).
```

may be transformed into clauses

```
member(X, Y) :-
        Y = [X | Y1]
    ?   true.
member(X, Y) :-
        Y = [X1 | Y1]
    ?   member(X, Y1).
```

where hiding is implicit according to (1). The equality constraints may then be folded back into the head according to (2), and the remaining null guards may be omitted in accordance with (3), giving

```
member(X, [X | Y1]) :-
    ?   true.
member(X, [X1 | Y1]) :-
    ?   member(X, Y1).
```

which may be further abbreviated to

```
member(X, [X | Y1]).
member(X, [X1 | Y1]) :-
        member(X, Y1).
```

according to (4) and (5).

We exemplify also with the append and merge definitions.

```
append([], Y, Z) :-
    → Y = Z.
append(X, Y, X) :-
    → X = [E | X1],
        Z = [E | Z1],
        append(X1, Y, Z1).
```

```
merge([], Y, Z) :-
    |   Y = Z.
merge(X, [], Z) :-
    |   X = Z.
merge([E | X], Y, Z) :-
    |   Z = [E | Z1],
        merge(X, Y, Z1).
merge(X, [E | Y], Z) :-
    |   Z = [E | Z1],
        merge(X, Y, Z1).
```

The examples should make it clear that some additional clarity is gained with the clausal syntax, which prevails in the logic programming community.

We end this section with a few additional remarks about the syntax.

As syntactic sugar, the underscore symbol "_" may be used in place of a variable that has a single occurrence in a clause. All occurrences of "_" in a definition denote different variables.

In an implementation of AKL, the character set restricts our syntax. The *then* symbol "→" is there written as "->", and subscripted indices are not possible. For example, append would be written as

```
append([], Y, Z) :-
   –>  Y = Z.
append(X, Y, Z) :-
   –>  X = [E | X1],
       append(X1, Y, Z1),
       Z = [E | Z1].
```

which is a program that can be compiled and run in AGENTS [Janson et al 1994]. However, to make programs as readable as possible, we will continue to use "→" and indices.

2.9 SUMMARY OF STATEMENTS

An informal summary of the different statements introduced follows.

Example	*Name*	*Section*
X = Y	Constraint atom	2.2, 2.3
append(X, Y, Z)	Program atom	2.3
p, q, r	Composition	2.3
X : p(X)	Hiding	2.3
(X = 0 → Y = 1 ; Y = 2)	Conditional choice	2.3, 2.6
(X = a | p(Z) ; Y = a | q(Z))	Committed choice	2.4
(X = 1 ? true ; X = 2 ? true)	Nondeterminate choice	2.5
bagof(X, member(X, [a,b,c]), L)	Bagof	2.7

A formal grammar is found in Chapter 4.

CHAPTER 3

PROGRAMMING PARADIGMS

Following the overview of AKL in the previous chapter, a number of representative examples illustrate the programming paradigms it provides.

3.1 A MULTIPARADIGM LANGUAGE

AKL is a *multiparadigm* programming language, supporting the following paradigms.

- processes and communication,
- object-oriented programming,
- functional and relational programming,
- constraint satisfaction.

These aspects of AKL are cleanly integrated, and provided using a minimum of basic concepts, common to them all. AKL agents will serve as processes, objects, functions, relations, or constraints, depending on the context (Figure 3.1).

Transformational aspects, such as the implicit search used for constraint satisfaction, co-exist harmoniously with *reactive* aspects, such as processes and process communication. Both may be used in the same program, with processes used for modelling the reactive interfaces to the user, external storage, and the outside world, and with a searching, don't know nondeterministic, component behaving as a process, encapsulating the nondeterminism.

As its name suggests, AKL is a programming language *kernel*. Some aspects of a complete programming language, a *user language*, have been omitted, such as type declarations and modules, a standard library, and direct syntactic support for some of the programming paradigms; but the *programming paradigms* and the basic *implementation technology* developed for AKL will carry over to any user language based on AKL.

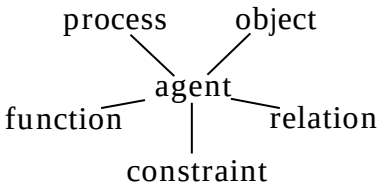

Figure 3.1. Multifaceted AKL agents

In the following sections, we will describe process programming in AKL, object-oriented programming in AKL, functional and relational programming in AKL, and constraint programming in AKL. Finally, it will be shown how these aspects may be integrated in an application.

3.2 PROCESSES AND COMMUNICATION

Agents may be thought of as processes, and telling constraints on shared variables may be thought of as communicating on a shared channel.

The basic principles supporting the idea of communicating processes were discussed in the AKL introduction (Section 2.3). Here we will expand the discussion by explaining many of the concurrent programming idioms. These are inherited from concurrent logic programming [Shapiro 1987; Taylor 1989; Tick 1991].

3.2.1 Communication and Streams

The underlying idea is that a logical variable may be used as a communication channel. On this channel, a message can be sent by a *producer* process by binding the variable to some value.

```
X = a
```

A conditional or a committed-choice statement may be used by a *consumer* process to achieve the effect of waiting for a message. By imposing suitable constraints on the communication variable in their guards, these statements will require the value of the variable to be defined before execution may proceed. Until the value has been produced, the statement will be suspended.

```
( X = a | this
; X = b | that )
```

However, as soon as the variable is constrained, the guard parts of these statements may be executed, and the appropriate action can be taken.

Message arguments can be transferred by binding the variable to a constructor expression.

```
X = f(Y)
```

Likewise, the argument can be received by matching against a constructor expression.

(Y : X = f(Y) | this(Y)
; Y : X = g(Y) | that(Y))

Again, note the scope of the hiding statement. It is limited to each guarded statement. If Y were given a wider scope, the first guard would instead be that the value of X should be equal to X = f(Y), for some given value of Y. The above use has the reading “if there exists a Y such that X = f(Y) …”, and it allows Y to be constrained by the guard.

Contrary to what is the case in the above examples, communication is not restricted to a single message between a producer and a consumer. A message can be given an argument that is the variable on which the next message will be sent. Usually, the list constructor is used for this purpose. The first argument of the list constructor is the message, and the second argument is the new variable. A sequence of messages M_i can be sent as follows.

$X_0 = [M_1 \mid X_1], X_1 = [M_2 \mid X_2], X_2 = [M_3 \mid X_3], \ldots$

The receiver waits for a list constructor, and expects the message to arrive in the first argument, and the variable on which further messages will be sent in the second argument. Observe that the above example is simply the construction of a list of messages. When used to transfer a sequence of messages between processes, a list is referred to as a *stream*. Just like a list, a stream may end with [], which indicates that the stream has been closed, and that no further messages will be sent.

Understood in these terms, the list-sum example in Section 2.3 is a typical producer-consumer example. The list agent produces a stream of messages, each of which is a number, and the sum agent consumes the stream, adding the numbers together.

3.2.2 Basic Stream Techniques

In the previous section, we discussed the notions of producers and consumers. The list-agent is an example of a producer, and the sum-agent is an example of a consumer. Further basic stream techniques are stream transducers, distributors, and mergers.

A stream *transducer* is an agent that takes one stream as input and produces another stream as output. This may involve computing new messages from old, rearranging, deleting, or adding messages. The following is a simple stream transducer computing the square of each incoming message.

```
squares([], Out) :-
    → Out = [].
squares([N | Ns], Out) :-
    → Out = [N*N | Out1],
      squares(Ns, Out1).
```

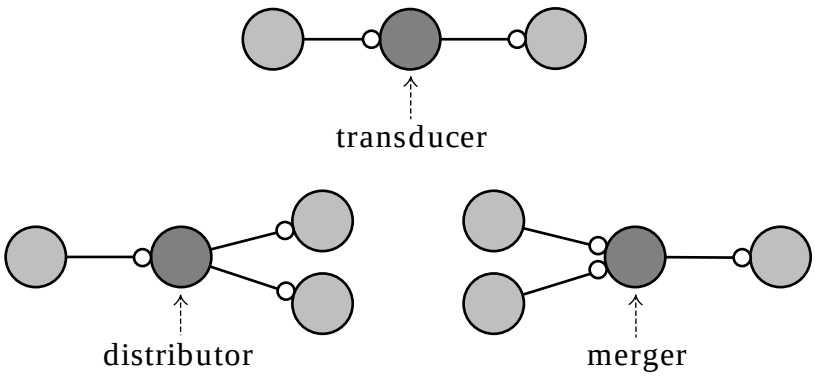

Figure 3.2. Transducer, distributor, and merger

A stream *distributor* is an agent with one input stream and several output streams that directs incoming messages to the appropriate output stream. The following is a simple stream distributor that sends apples to one stream and oranges to the other.

```
fruits([], As, Os) :-
    → As = [],
      Os = [].
fruits([F | Fs], As, Os) :-
      apple(F)
    → As = [F | As1],
      fruits(Fs, As1, Os).
fruits([F | Fs], As, Os) :-
      orange(F)
    → Os = [F | Os1],
      fruits(Fs, As, Os1).
```

A stream *merger* is an agent with several input streams and one output stream that interleaves messages from the input streams into the single output stream. The following is the standard binary stream merger, which was also shown in the language introduction (Section 2.4).

```
merge([], Ys, Zs) :-
    |  Zs = Ys.
merge(Xs, [], Zs) :-
    |  Zs = Xs.
merge([X | Xs], Ys, Zs) :-
    |  Zs = [X | Zs1],
       merge(Xs, Ys, Zs1).
merge(Xs, [Y | Ys], Zs) :-
    |  Zs = [Y | Zs1],
       merge(Xs, Ys, Zs1).
```

Note that all the above definitions can also be seen as simple list-processing agents. However, they are more interesting when one considers their behaviour as components in concurrent programs.

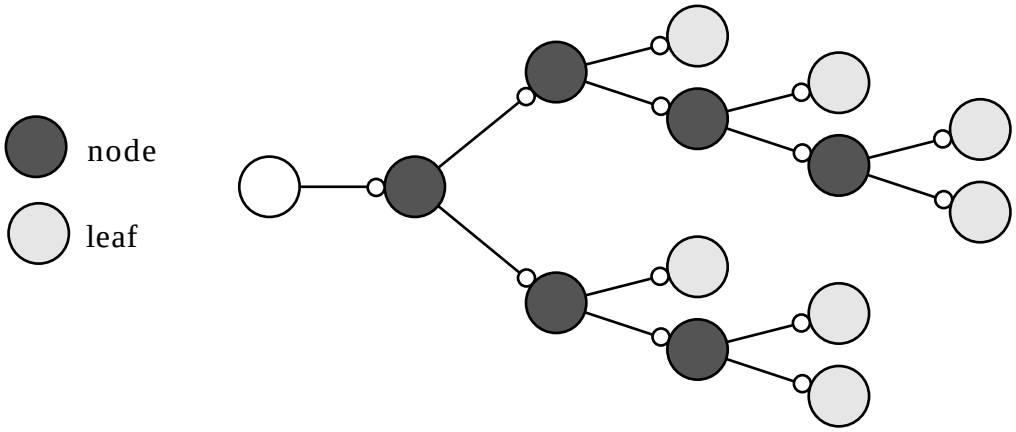

Figure 3.3. Tree of node and leaf processes

3.2.3 Process Structures

Process networks is a technique for storing data, and an example of an object-oriented reading of processes. The technique is best introduced by an example. We will show how a dictionary can be represented as a binary tree of processes.

The tree is built from leaf processes and node processes. A leaf process has one input stream from its parent. A node process has one input stream from its parent and two output streams to its children. In addition, it has two arguments for holding the key and the value of the data item stored in the node. Thus, the processes correspond to equivalent data-structures.

In their default state, these processes are waiting for messages on their input streams. The messages may be of the kind insert(Key, Value), with given key and value that should be inserted, lookup(Key, Result), with a given key and a sought for result (an unconstrained variable), and the closing of the stream which means that the tree should terminate (deallocate itself). Messages that include a variable for the return value are called *incomplete messages*.

The computed result is wrapped in the constructor found(Value), if a value corresponding to a key is found, and is otherwise simply the constant “not_found”.

When a node process receives a request, it compares the key to the key held in its argument, and either takes care of the request itself, or passes the request along to its left or right sub-tree, depending on the result of the comparison. A leaf process always processes a request itself.