An Introduction to AKL
A Multi-Paradigm Programming Language*

Sverker Janson and Seif Haridi
Swedish Institute of Computer Science
Box 1263, S-164 28 KISTA, Sweden
E-mail sverker@sics.se, seif@sics.se

December 14, 1993

1 Introduction

AKL is a multi-paradigm programming language based on a concurrent constraint framework [3], directly or indirectly supporting the following paradigms.

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

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.

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.

In the following sections, we will introduce AKL, then 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.

2 Concurrent Constraint Programming

AKL is based on the concept of concurrent constraint programming, a paradigm distinguished by its elegant notions of communication and synchronisation based on constraints [7].

*Also in NATO-ASI Constraint Programming, Springer-Verlag, forthcoming

In a concurrent constraint programming language, a computation state consists of a group of *agents* and a *store* that they share. Agents may add pieces of information to the store, an operation called *telling*, and may also wait for the presence in the store of pieces of information, an operation called *asking*. The information in the store is expressed in terms of *constraints*, which are statements in some constraint language, usually based on first-order logic, e.g.,

$$X < 1,\ Y = Z + X,\ W = [a, b, c],\ \ldots$$

If telling makes a store inconsistent, the computation fails (more on this later). Asking a constraint means waiting until the asked constraint either is *entailed* by (follows logically from) the information accumulated in the store or is *disentailed* by (the negation follows logically from) the same information. In other words, no action is taken until it has been established that the asked constraint is true or false. For example, $X < 1$ is obviously entailed by $X = 0$ and disentailed by $X = 1$.

Constraints restrict the range of possible values of variables that are shared between agents. A variable may be thought of as a container. Whereas variables in conventional languages hold single values, variables in concurrent constraint programming languages may be thought of as holding the (possibly infinite) set of values consistent with the constraints currently in the store. This extensional view may be complemented by an intensional view, in which each variable is thought of as holding the constraints which restrict it. This latter view is often more useful as a mental model.

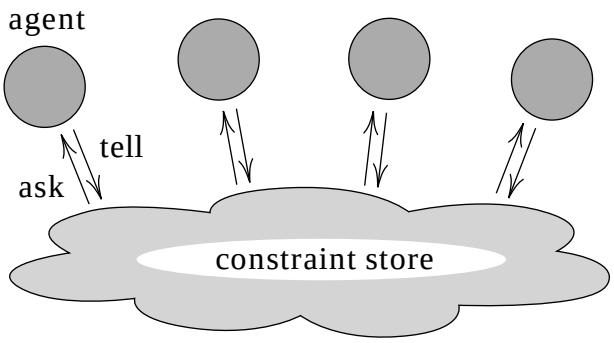

Figure 1: Agents interacting with a constraint store

The range of constraints that may be used in a program is defined by the current *constraint system*, which in AKL, in principle, may be any first-order theory. In practice, it is necessary to ensure that the telling and asking operations used are computable and have a reasonable computational complexity. Constraint systems as such are not discussed here. For the purpose of this introduction, we will use a simple constraint system with a few obvious constraints, which is essentially that of Prolog and GHC to which arithmetic has been added.

Thus, constraints in AKL will be formulas of the form

$$\langle expression \rangle = \langle expression \rangle$$
$$\langle expression \rangle \neq \langle expression \rangle$$
$$\langle expression \rangle < \langle expression \rangle$$

and the like. Equality constraints, e.g., X = 1, are often called *bindings*, suggesting that the variable X is *bound* to 1 by the constraint. Correspondingly, the act of telling a binding on a variable is called *binding* the variable. Expressions are either *variables* (alpha-numeric symbols with an upper case initial letter), e.g.,

$$X, Y, Z, X_1, Y_1, Z_1, \ldots$$

or *numbers*, e.g.,

$$1, 3.1415, -42, \ldots$$

or *arithmetic expressions*, e.g.,

$$1 + X, -Y, X * Y, \ldots$$

or *constants*, e.g.,

$$a, b, c, \ldots$$

or *constructor expressions* of the form

$$\langle name \rangle(\langle expression \rangle, \ldots, \langle expression \rangle)$$

where $\langle name \rangle$ is an alpha-numeric symbol with a lower case initial letter, e.g.,

$$s(s(0)), tree(X, L, R), \ldots$$

There is also the constant [], which denotes the empty list, and the list constructor $[\langle expression \rangle | \langle expression \rangle]$. A syntactic convention used in the following is that, e.g., the expression [a|[b|[c|d]]] may be written as [a, b, c|d], and the expression [a|[b|[c|[]]]] may be written as [a, b, c]. In addition we assume that constraints *true* and *false* are available, which are independent of the constraint system and may be identified with their corresponding logical constants.

3 Basic Concepts

The agents of concurrent constraint programming correspond to statements being executed concurrently. Constraints, as described in the previous section, are atomic statements known as *constraint atoms* (or just constraints). When they are asked and when they are told is discussed in the following.

A *program atom* statement of the form

$$\langle name \rangle(X_1, \ldots, X_n)$$

is a defined agent. In a program atom, $\langle name \rangle$ is an alpha-numeric symbol and n is the arity of the atom. The variables $X_1, \ldots, X_n$ are the *actual parameters* of the atom. Occurrences of program atoms in programs are sometimes referred to as *calls*. Atoms of the above form may be referred to as $\langle name \rangle / n$ atoms, e.g.,

$$plus(X, Y, Z)$$

is a plus/3 atom. Occasionally, when no ambiguity can arise, "$/n$" is dropped. The behaviour of atoms is given by *(agent) definitions* of the form

$$\langle name \rangle(\mathrm{X}_1, \ldots, \mathrm{X}_n) := \langle statement \rangle.$$

The variables $\mathrm{X}_1, \ldots, \mathrm{X}_n$ must be different and are called *formal parameters*. During execution, any atom matching the left hand side will be replaced by the statement on the right hand side, with actual parameters replacing occurrences of the formal parameters. A definition of the above form is said to define the $\langle name \rangle / n$ atom, e.g.,

$$\mathrm{plus}(\mathrm{X}, \mathrm{Y}, \mathrm{Z}) := \mathrm{Z} = \mathrm{X} + \mathrm{Y}.$$

is a definition of plus/3.

A *composition* statement of the form

$$\langle statement \rangle, \ldots, \langle statement \rangle$$

builds a composite agent from a sequence of agents. Its behaviour is to replace itself with the concurrently executing agents corresponding to its components.

A *conditional choice* statement of the form

$$(\langle statement \rangle \rightarrow \langle statement \rangle \ ; \langle statement \rangle)$$

is used to express conditional execution. Let us call its components condition, then-branch, and else-branch, respectively. (Later a more general version of this statement will be introduced.)

Let us, for simplicity, assume that the condition is a constraint. A conditional choice statement will ask the constraint in the condition from the store. If it is entailed, the then-branch replaces the statement. If it is disentailed, the else-branch replaces the statement. If neither, the statement will wait until either becomes known. If the condition is an arbitrary statement, the above described actions will take place when the condition has been reduced to a constraint or when it fails. The concept of failure is discussed later.

A *hiding* statement of the form

$$\mathrm{X}_1, \ldots, \mathrm{X}_n : \langle statement \rangle$$

introduces variables with local scope. The behaviour of a hiding statement is to replace itself with its component statement, in which the variables $\mathrm{X}_1, \ldots, \mathrm{X}_n$ have been replaced by new variables.

Let us at this point establish some syntactic conventions.

- Composition binds tighter than hiding, e.g.,

  $$\mathrm{X} : \mathrm{p}, \mathrm{q}, \mathrm{r}$$

  means

  $$\mathrm{X} : (\mathrm{p}, \mathrm{q}, \mathrm{r})$$

  Parentheses may be used to override this default, e.g.,

  $$(\mathrm{X} : \mathrm{p}), \mathrm{q}, \mathrm{r}$$

- Any variable occurring free in a definition (i.e., not as one of the formal parameters, nor introduced by a hiding statement) is implicitly introduced by a hiding statement enclosing the right hand side of the definition, e.g.,

  $$p(X, Y) := q(X, Z), r(Z, Y).$$

  where Z occurs free, means

  $$p(X, Y) := Z : q(X, Z), r(Z, Y).$$

  in which hiding has been made explicit.

- Expressions may be used as arguments to program atoms, and will then correspond to bindings on the actual parameters, e.g.,

  $$p(X+1, [a, b, c])$$

  means

  $$(Y, Z : Y = X+1, Z = [a, b, c], p(Y, Z))$$

  where the new arguments have also been made local by hiding.

It is now time for a first small example: an append/3 agent which is used to concatenate two lists.

$$\begin{aligned} &append(X, Y, Z) := \\ &\quad (X = [] \rightarrow Z = Y \\ &\quad ; X = [E|X_1], append(X_1, Y, Z_1), Z = [E|Z_1]). \end{aligned}$$

It will initially suffice to think about constraints in two different ways, depending on the context in which they occur. When occurring as conditions, constraints are asked. Elsewhere, they are told.

In append/3, the condition $X = []$ is asked, which means that it may be read "as usual". If it is entailed, the then-branch is chosen, in which $Z = Y$ is told. If the condition is disentailed, the else-branch is chosen. There, $X = [E|X_1]$ is told. Since X is not [], it is assumed that it is a list constructor, in which E is equal to the head of X and X_1 equal to the tail of X. The recursive append call makes Z_1 the concatenation of X_1 and Y. The final constraint $Z = [E|Z_1]$ builds the output Z from E and the partial result Z_1.

Note how variables allow us to work with incomplete data. In a call

$$append([1, 2, 3], Y, Z)$$

the parameters Z and Y can be left unconstrained. The third parameter Z may still be computed as $[1, 2, 3|Y]$, where the tail Y is unconstrained. If Y is later constrained by, e.g., $Y = []$, then it is also the case that $Z = [1, 2, 3]$.

Variables are also indirectly the means of communication and synchronisation. If a constraint on a variable is asked, the corresponding agent, e.g., conditional choice statement, is suspended and may be restarted whenever an appropriate constraint is told on the variable by another agent.

At this point it seems appropriate to illustrate the nature of concurrent computation in AKL. The following definitions will create a list of numbers, and add together a list of numbers, respectively.

$\mathrm{list}(N, L) :=$
$\quad (\ N = 0 \rightarrow L = [] \ ; \ L = [N|L_1], \mathrm{list}(N - 1, L_1)\ ).$

$\mathrm{sum}(L, N) :=$
$\quad (\ L = [] \rightarrow N = 0 \ ; \ L = [M|L_1], \mathrm{sum}(L_1, N_1), N = N_1 + M\ ).$

The following computation is possible. In the examples, computations will be shown by performing rewriting steps on the state (or configuration) at hand, unfolding definitions and substituting values for variables, etc., where appropriate, which should be intuitive. In this example we avoid details by showing only the relevant atoms and the collection of constraints on the output variable N. Intermediate computation steps are skipped. Thus,

$$\mathrm{list}(3, L), \mathrm{sum}(L, N)$$

is rewritten to

$$\mathrm{list}(2, L_1), \mathrm{sum}([3|L_1], N)$$

by unfolding the list atom, executing the choice statement, and substituting values for variables according to equality constraints. This result may in its turn be rewritten to

$$\mathrm{list}(1, L_2), \mathrm{sum}([2|L_2], N_1), N = 3 + N_1$$

by similar manipulations of the list and sum atoms. Further possible states are

$$\mathrm{list}(0, L_3), \mathrm{sum}([1|L_3], N_2), N = 5 + N_2$$
$$\mathrm{sum}([], N_3), N = 6 + N_3$$
$$N = 6$$

with final state $N = 6$.

The list/2 call produces a list, and the sum/2 call is there to consume its parts as soon as they are created. The logical variable allows the sum/2 call to know when data has arrived. If the tail of the list being consumed by the sum/2 call is unconstrained, the sum/2 call will wait for it to be produced (in this case by the list/2 call).

In this example, a particular execution order was chosen, but observe that the final result is quite independent of the execution order.

The simple set of constructs introduced so far is a fairly complete programming language in itself, quite comparable in expressive power to, e.g., functional programming languages. If we were merely looking for Turing completeness, the language could be restricted, and the constraint systems could be weakened considerably. But then important aspects such as concurrency, modularity, and, of course, expressiveness would all be sacrificed on the altar of simplicity.

In the following sections, we will introduce constructs that address the specific needs of important programming paradigms, such as processes and process communication, object-oriented programming, relational programming, and constraint satisfaction. In particular, we will need the ability to choose between alternative computations in a manner more flexible than that provided by conditional choice.

4 Don't Care Nondeterminism

In concurrent programming, processes should be able to react to incoming communication from different sources. In constraint programming, constraint propagating agents should be able to react to different conditions. Both of these cases can be expressed as a number of possibly non-exclusive conditions with corresponding branches. If one condition is satisfied, its branch is chosen.

For this, AKL provides the *committed choice* statement

$$
\begin{array}{l}
(\ \langle \mathit{statement} \rangle \mid \langle \mathit{statement} \rangle \\
;\ \ldots \\
;\ \langle \mathit{statement} \rangle \mid \langle \mathit{statement} \rangle\ )
\end{array}
$$

The symbol "|" is called *commit*. The statement preceding commit is called a *guard* and the statement following it is called a *body*. A pair

$$\langle \mathit{statement} \rangle \mid \langle \mathit{statement} \rangle$$

is called a *(guarded) clause*, and may be enclosed in hiding as follows.

$$X_1, \ldots, X_n : \langle \mathit{statement} \rangle \mid \langle \mathit{statement} \rangle$$

The variables $X_1, \ldots, X_n$ are called *local variables* of the clause.

Let us first, for simplicity, assume that the guards are all constraints. The committed-choice statement will ask all guards from the store. If any of the guards is entailed, the composition of its constraint and its corresponding body replaces the committed-choice statement. If a guard is disentailed, its corresponding clause is deleted. If all clauses are deleted, the committed choice statement fails. Otherwise, it will wait. Thus, it may select an arbitrary entailed guard, and commit the computation to its corresponding body.

If a variable Y is hidden, an asked constraint is preceded by the expression "for some Y" (or logically, "$\exists$Y"). For example, in

$$X = f(a),\ (\ Y : X = f(Y) \mid q(Y)\ )$$

the asked constraint is $\exists Y(X = f(Y))$ ("for some Y, X = f(Y)"), which is entailed, since there exists a Y (namely "a") such that X = f(Y) is entailed.

List merging may now be expressed as follows, as an example of an agent receiving input from two different sources.

$$
\begin{array}{l}
\mathrm{merge}(X, Y, Z) := \\
\qquad (\ X = [] \mid Z = Y \\
\qquad ;\ Y = [] \mid Z = X \\
\qquad ;\ E, X_1 : X = [E|X_1] \mid Z = [E|Z_1],\ \mathrm{merge}(X_1, Y, Z_1) \\
\qquad ;\ E, Y_1 : Y = [E|Y_1] \mid Z = [E|Z_1],\ \mathrm{merge}(X, Y_1, Z_1)\ ).
\end{array}
$$

A merge agent can react as soon as either X or Y is given a value. In the last two clauses, hiding introduces variables that are used for "matching" in the guard, as discussed above. These variables are constrained to be equal to the corresponding list components.

5 Don't Know Nondeterminism

Many problems, especially frequent in the field of Artifical Intelligence, and also found elsewhere, e.g., in operations research, are currently solvable only by resorting to some form of search. Many of these admit very concise solutions if the programming language abstracts away the details of search by providing don't know nondeterminism.

For this, AKL provides the *nondeterminate choice* (or *don't know choice*) statement.

(⟨*statement*⟩ ? ⟨*statement*⟩
; . . .
; ⟨*statement*⟩ ? ⟨*statement*⟩)

The symbol "?" is called *wait*. The statement is otherwise like the committed choice statement in that its components are called *(guarded) clauses*, the components of a clause *guard* and *body*, and a clause may be enclosed in hiding.

Again we assume that the guards are all constraints. The nondeterminate choice statement will also ask all guards from the store. If a guard is disentailed, its corresponding clause is deleted. If all clauses are deleted, the choice statement fails. If only one clause remains, the choice statement is said to be determinate. Then the composition of the constraint in the remaining guard and its corresponding body replaces the choice statement. Otherwise, if there is more than one clause left, the choice statement will wait. Subsequent telling of other agents may make it determinate. If eventually a state is reached in which no other computation step is possible, each of the remaining clauses may be tried in different copies of the state. The alternative computation paths are explored concurrently.

Let us first consider a very simple example, an agent that accepts either of the constants a or b, and then does nothing.

```
p(X) :=
        ( X = a ? true
        ; X = b ? true ).
```

The interesting thing happens when the agent p is called with an unconstrained variable as an argument. That is, we expect it to produce output. Let us call p together with an agent q examining the output of p.

```
q(X, Y) :=
        ( X = a → Y = 1
        ; Y = 0 ).
```

Then the following is one possible computation starting from

p(X), q(X, Y)

First p and q are both unfolded.

(X = a ? true ; X = b ? true), (X = a → Y = 1 ; Y = 0)

At this point in the computation, the nondeterminate choice statement is nondeterminate, and the conditional choice statement cannot establish the truth or falsity of its condition. The computation can now only proceed by trying the clauses of the nondeterminate choice in different copies of the computation state. Thus,

X = a, (X = a → Y = 1 ; Y = 0)
Y = 1

and

X = b, (X = a → Y = 1 ; Y = 0)
Y = 0

are the two possible computations. Observe that the nondeterminate alternatives are ordered in the order of the clauses in the nondeterminate choice statement. This ordering will be used later.

Now, what could possibly be the use of having an agent generate alternative results? This we will try to answer in the following. It will help to think of the alternative results as a sequence of results. Composition of two agents will compute the intersection of the two sequences of results. This will be illustrated using the member agent, which examines membership in a list.

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

The agent

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

will establish whether the value of X is in the list [a, b, c]. When the agent is called with an unconstrained X, the different members of the list are returned as different possible results (in the order a, b, c, due to the way the program is written). The composition

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

will compute the X that are members in both lists. When two nondeterminate choice statements are available, the leftmost is chosen. In this case it will enumerate members of the first list, creating three alternative states

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

The members in the first list that are not members in the second are eliminated by the failure of the corresponding alternative computations. A computation that fails leaves no trace in the sequence of results, and the two final alternative states will be

X = b
X = c

In fact, the sequence of results may become empty, as in the case of the following composition

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

Such complete failure is also useful, as discussed in the following.

6 General Statements in Guards

Although we have ignored it up to this point, any statement may be used as a guard in a choice statement. The behaviour presented above has been that of the special case when conditions and guards are constraints. This will now be generalised.

Before we proceed, we introduce the general conditional choice statement.

$$
\begin{array}{l}
(\ \langle \mathit{statement} \rangle \rightarrow \langle \mathit{statement} \rangle \\
;\ \ldots \\
;\ \langle \mathit{statement} \rangle \rightarrow \langle \mathit{statement} \rangle\ )
\end{array}
$$

The symbol "$\rightarrow$" is called *then*. Again, the statement is otherwise like the other choice statements in that its components are called *(guarded) clauses*, the components of a clause *guard* and *body*, and a clause may be enclosed in hiding.

The previously introduced version of conditional choice is, of course, merely syntactic sugar for the special case

$$
\begin{array}{l}
(\ \langle \mathit{statement} \rangle \rightarrow \langle \mathit{statement} \rangle \\
;\ \mathrm{true} \rightarrow \langle \mathit{statement} \rangle\ )
\end{array}
$$

The case where the guard of the last clause is "true" is common enough to warrant general syntactic sugar, thus

$$
\begin{array}{l}
(\ \langle \mathit{statement} \rangle \rightarrow \langle \mathit{statement} \rangle \\
;\ \ldots \\
;\ \mathrm{true} \rightarrow \langle \mathit{statement} \rangle\ )
\end{array}
$$

may always be abbreviated to

$$
\begin{array}{l}
(\ \langle \mathit{statement} \rangle \rightarrow \langle \mathit{statement} \rangle \\
;\ \ldots \\
;\ \langle \mathit{statement} \rangle\ )
\end{array}
$$

For the last time we make the simplifying assumption that the guards are all constraints. The conditional choice statement asks the constraint of the first guard. If it is entailed, the composition of it and its body replaces the choice statement. If it is disentailed, the clause is deleted, and the next clause is tried. If neither, the statement will wait. These steps are repeated as necessary. If no clauses remain, the conditional choice statement fails.

When a more general statement is used as a guard, it will first be executed locally in the guard, reducing itself to a constraint, after which the previously described actions take place. To illustrate this before we descend into the details, let us use append in a guard (a fairly unusual guard though).

$$
\begin{array}{l}
(\ \mathrm{append(X,\ Y,\ Z)} \rightarrow \mathrm{p(Z)} \\
;\ \mathrm{true} \rightarrow \mathrm{q(X,\ Y)}\ )
\end{array}
$$

If we supply constraints for X and Y, e.g., X = [1], Y = [2,3], a value will be computed locally for Z, and the resulting choice statement is

$$
\begin{array}{l}
(\ \mathrm{Z} = [1,2,3] \rightarrow \mathrm{p(Z)} \\
;\ \mathrm{true} \rightarrow \mathrm{q(X,\ Y)}\ )
\end{array}
$$

with its above described behaviour.

Formally, the computation in the guard is a separate computation, with local agents and its own local constraint store. Constraints told by local agents are placed in the local store, but constraints asked by local agents are asked from the union of the local store and external stores. Locally told constraints can thus be observed by local agents, but not by agents external to the guard.

When the local computation terminates successfully, the constraint asked for the guard is the conjunction of constraints in its local constraint store. This coincides with the behaviour in the special case that the guard was a constraint. In fact, the behaviour of a constraint atom statement is always to tell its constraint to the current constraint store.

If the local store becomes inconsistent with the union of external stores, the local computation fails. The behaviour is then as if the computation had terminated successfully, its constraint had been asked, and it had been found disentailed by the external stores.

The scope of don't know nondeterminism in a guard is limited to its corresponding clause. New alternative computations for a guard will be introduced as new alternative clauses. This will be illustrated using the following simple nondeterminate agent.

```
one_or_one(X, Y) :=
        ( X = 1 ? true
        ; Y = 1 ? true ).
```

Let us start with the statement

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

The one_or_one 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 exactly when alternatives may be tried. A (possibly local) state with agents and their store is *(locally) stable* if no computation step other than copying in nondeterminate choice is possible, 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

$$(Y_1 : Y = [X|Y_1] \ ? \ \text{true}$$
$$; X_1, Y_1 : Y = [X_1|Y_1] \ ? \ \text{member}(X, Y_1))$$

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.

7 Bagof

Finally, we introduce a statement which builds lists of 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., Haskell).

A *bagof* statement of the form

$$\text{bagof}(\langle \textit{variable} \rangle, \langle \textit{statement} \rangle, \langle \textit{variable} \rangle)$$

builds a list of the sequence of alternative results from its component statement. The different alternative bindings 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. Don't know nondeterminism is not propagated outside it.

For example, the composition

$$\text{member}(X, [a, b, c]), \text{member}(X, [b, c, d])$$

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

$$\text{bagof}(X, (\text{member}(X, [a, b, c]), \text{member}(X, [b, c, d])), Y)$$

the result becomes

$$Y = [b, c]$$

as could be expected. 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,

$$\text{bagof}(X, ((X = a \ ; X = b) \ ; (X = c \ ; X = d)), Y)$$

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

$$\text{unordered\_bagof}(X, ((X = a \ ; X = b) \ ; (X = c \ ; X = d)), Y)$$

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].

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

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

where % is either $\rightarrow$, |, or ?, may be broken up into several clauses

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

which together stand for the above definition.

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(X1, ..., Xn) of these clauses. (3) If the remainder of the guard is the null statement "true", it may be omitted. (4) If the guard is omitted and the guard operator is wait "?", it may also be omitted. (5) If the guard operator is omitted, and the body is the null statement "true", a clause may be abbreviated to a head.

As an example, the definition

member(X, Y) :=
$\quad (Y_1 : Y = [X|Y_1] \ ? \ \text{true}$
$\quad ; X_1, Y_1 : Y = [X_1|Y_1] \ ? \ \text{member}(X, Y1) \ ).$

may be transformed to clauses

member(X, Y) :–
$\quad Y = [X|Y_1]$
? $\quad$ true.
member(X, Y) :–
$\quad Y = [X_1|Y_1]$
? $\quad \text{member}(X, Y_1).$

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 according to (3), giving

member(X, [X|Y_1]) :–
? $\quad$ true.

member(X, [X_1|Y_1]) :–
? $\quad \text{member}(X, Y_1).$

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 the AKL Programming System [5]. However, to make programs as readable as possible, we will continue to use "→" and indices.

9 Processes and Process 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 previous sections. Here we will expand the discussion by explaining many of the concurrent programming idioms. These are inherited from concurrent logic programming (see, e.g., [8]).

9.1 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.

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

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 above 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.

9.2 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.

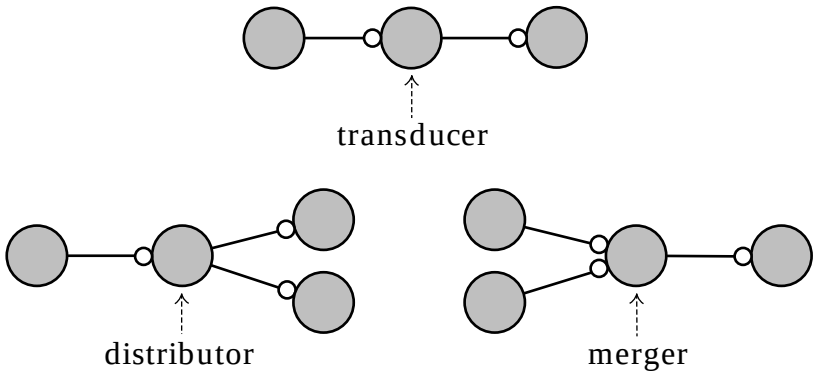

Figure 2: Transducer, distributor, and merger

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).
```

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.

```
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.

9.3 Process Structures

Process networks can be used for storing data. This is 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.

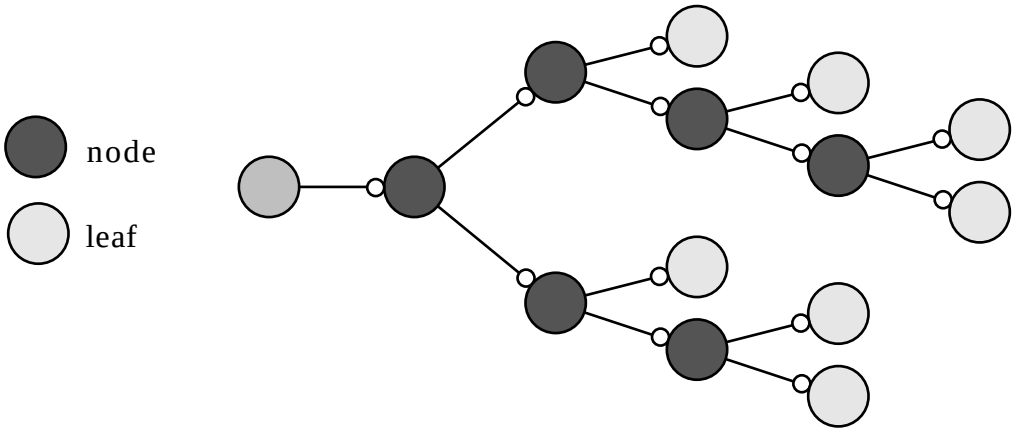

Figure 3: Tree of node and leaf processes

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).

This technique, to include a variable in the message for the return value, is common enough to warrant a name of its own: *incomplete messages*.

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

```
dict(S) := leaf(S).

leaf([]) :-
        →       true.
leaf([insert(K,V)|S]) :-
        →       node(S, K, V, L, R),
                leaf(L),
                leaf(R).
leaf([lookup(K,V)|S]) :-
        →       V = not_found,
                leaf(S).

node([], _, _, L, R) :-
        →       L = [],
                R = [].
node([insert(K1, V1)|S], K, V, L, R) :-
        →       (               K1 = K
                        →       node(S, K, V1, L, R)
                ;               K1 < K
                        →       L = [insert(K1, V1)|L1],
                                node(S, K, V, L1, R)
                ;               K1 > K
                        →       R = [insert(K1, V1)|R1],
                                node(S, K, V, L, R1) ).
node([lookup(K1, V1)|S], K, V, L, R) :-
        →       (               K1 = K
                        →       V1 = found(V),
                                node(S, K, V, L, R)
                ;               K1 < K
                        →       L = [lookup(K1, V1)|L1],
                                node(S, K, V, L1, R)
                ;               K1 > K
                        →       R = [lookup(K1, V1)|R1],
                                node(S, K, V, L, R1) ).
```

In the following section on object-oriented programming, we will relate this programming technique to conventional object-oriented programming and its standard terminology.

10 Object-Oriented Programming

In this section, the basic techniques that allow us to do object-oriented programming in AKL are reviewed. Like the programming techniques in the previous section, they belong to logic programming folklore.

There is more than one way to map the abstract concept of an object onto corresponding concepts in a concurrent constraint language. The first and most widespread of these will be described here in detail [9]. It is based on the process reading of logic programs. Several embedded languages have been proposed that support this style of programming (e.g., [6, 11, 1]). They are typically much less verbose, and they also provide more explicit support for objectoriented concepts.

As will be seen, in this framework there is no real need for an implementation of objects, unlike the case when one is adding object oriented support to a language such as C. Following an object-oriented style of programming is a very natural thing.

10.1 Objects

An *object* is an abstract entity that provides services to its clients. Clients explicitly request services from objects. The request identifies the requested service, as well as the objects that are to perform the service.

Objects are realised as processes that take as input a stream (a list) of requests. The stream identifies the object. The data associated with the objects are held in the arguments of the process. An object definition typically has one clause per type of request, which performs the corresponding service, and one clause for terminating (or deallocating) the object. Thus, clauses correspond to methods.

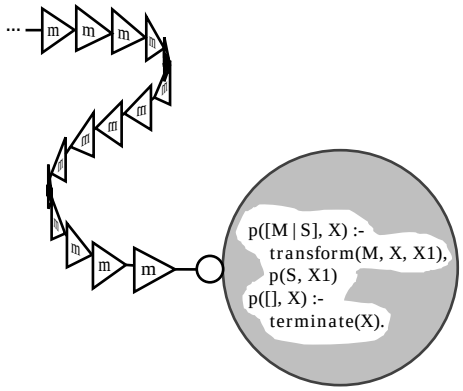

Figure 4: An object consuming a list of messages

The requests are typically expressions of the form name(A, B, C), where the constructor "name" identifies the request, and A, B, and C are the arguments of the request.

The process description, the agent definition, is the class, the implementation of the object. The individual calls to this agent are the instances. A standard example of an object is the bank account, providing withdrawal, deposits, etc.

```
make_bank_account(S) :=
        bank_account(S, 0).
```

```
bank_account([], _) :-
      →    true.
bank_account([withdraw(A)|R], N) :-
      →    bank_account(R, N − A).
bank_account([deposit(A)|R], N) :-
      →    bank_account(R, N + A).
bank_account([balance(M)|R], N) :-
      →    M = N,
           bank_account(R, N).
```

A computation starting with

$$\text{make\_bank\_account}(S),\quad S = [\text{balance}(B_1), \text{deposit}(7), \text{withdraw}(3), \text{balance}(B_2)]$$

yields

$$B_1 = 0,\ B_2 = 4$$

A bank-account object is created by starting a process bank_account(S, 0) given as initial input an unspecified stream S (a variable) and a zero balance. The stream S is used to identify the object. A service deposit(5) is requested by binding S to [deposit(5)|S_1]. The next request is added to S_1, and so on. In the above example, only one clause will match any given request. When it is applied, some computation is performed in its body and a new bank_account process replaces the original one. The requests in the above example are processed as follows. Let us start in the middle.

$$\text{bank\_account}(S, 0),\ S = [\text{deposit}(7), \text{withdraw}(3), \text{balance}(B_2)].$$

The bank_account process is reduced by the clause matching the first deposit request, leaving some computation to be performed.

$$N = 0+7,\ \text{bank\_account}(S_1, N),\ S_1 = [\text{withdraw}(3), \text{balance}(B_2)].$$

This leaves us with.

$$\text{bank\_account}(S, 7),\ S_1 = [\text{withdraw}(3), \text{balance}(B_2)].$$

The rest of the requests are processed similarly.

Finally, there are a few things to note about these objects. First, they are automatically encapsulated. Clients are prevented from directly accessing the data associated with an object. In imperative languages, this is not as self-evident, as the object is often confused with the storage used to store its internal data, and the object identifier is a pointer to this storage, which may often be used for any purpose.

Second, requests are entirely generic. The expression that identifies a request may be interpreted differently, and may therefore involve the execution of different code, depending on the object. This does not involve mandatory declarations in some shared (abstract or virtual) ancestor class, as in many other languages.

Third, becoming another type of object is extremely simple. Instead of replacing itself with an object of the same type, an object may pass its stream, and appropriate parameters, on to a new object. An example of this was given in the section on process structures, where a leaf process became a node process when a message was inserted into a binary tree.

10.2 Inheritance

In the object-oriented paradigm, objects can be classified in terms of the services they provide. One object may provide a subset of the services of another object. This way an interface hierarchy is formed.

It is of course important, from a software engineering point of view, that the descriptions of objects higher up in the hierarchy can be reused as parts of the descendant objects. This is either done by inheritance or by delegation. Delegation is easily achieved in the framework we describe. However, since requests are completely generic, it is also possible to design an interface hierarchy without inheritance or delegation, if so desired.

Delegation is achieved by creating instances of the ancestor objects. The object identifier of (the stream to) this ancestor object is held as an argument of the derived object. The object corresponding to the ancestor could appropriately be called a subobject of the derived object. The derived object filters incoming requests and delegates unknown requests to its subobject.

Delegation is not restricted to unknown requests. We may also define what is elsewhere known as after- and before-methods by filtering as well. The derived object may perform any action before passing a request on to a subobject.

Let us derive from the bank_account class a kind of account that does some form of logging of incoming requests. Let us say that it also adds a get_log service that returns the log. This is easy.

```
make_logging_account(S) :=
        make_bank_account(O),
        make_empty_log(Log),
        logging_account(S, O, Log).

logging_account([get_log(L)|R], O, Log) :-
    →   L = Log,
        logging_account(R, O, Log).
logging_account([Req|R], O, Log) :-
    →   O = [Req|O1],
        add_to_log(Req, Log, Log1),
        logging_account(R, O1, Log1).
logging_account([], O, _) :-
    →   O = [].
```

With delegation, it is cumbersome to handle the notion of self correctly. Modern forms of multiple inheritance, based on the principle of specialisation, are also difficult to achieve. Instead, it is quite possible to view inheritance as providing the ability to share common portions of object definitions by placing them in superclasses, which are then implicitly copied into sub-class definitions. To exploit this view, syntactic support has to be added to the language, e.g., along the lines of Goldberg and Shapiro [2]. This view corresponds closely to that of conventional object-oriented languages.

10.3 Ports for Objects

Ports are a special form of constraints, which, when added to AKL, or to any concurrent logic programming language, will solve a number of problems with the

approach to object-oriented programming presented above, problems that we have avoided mentioning so far. This section provides a preliminary introduction to ports. They, and the problems they solve, are described in great detail elsewhere [4].

A port is a binary constraint on a bag (a multi-set) of messages and a corresponding stream of these messages. It simply states that they contain the same messages, in any order. A bag connected to a stream by a port is usually identified with the port, and is referred to as a port. The open_port(P, S) operation relates a bag P to a stream S, and connects them through a port. The stream S will usually be connected to an object. Instead of using the stream to access the object, we will send messages by adding them to the port. The send(M, P) operation sends a message M to a port P. To satisfy the port constraint, a message sent to a port will immediately be added to its associated stream, first come first served.

When a port is no longer referenced from other parts of the computation state, when it becomes garbage, it is assumed that it contains no more messages, and its associated stream is automatically closed. When the stream is closed, any object consuming it is thereby notified that there are no more clients requesting its services.

Thus, to summarise: A port is created with an associated stream (to an object). Messages are sent to the port, and appear on the stream in any order. When the port is no longer in use, the stream is closed, and the object may choose to terminate.

A simple example follows.

open_port(P, S), send(a, P), send(b, P)

yields

P = $\langle a\ port \rangle$, S = [a, b]

Here we create a port and a related stream, and send two messages. The messages appear in S in the order of the send operations in the composition, but it could just as well have been reversed. The stream is closed when the messages have been sent, since there are no more references to the port.

Ports solve a number of problems that are implicit in the use of streams. The following are the most obvious.

- If several clients are to access the same object, their streams of messages have to be merged into a single input stream. With ports, no merger has to be created. Any client can send a message on the same port.
- If objects are to be embedded in other data structures, creating, e.g., an array of objects, streams have to be put in these structures. Such structures cannot be shared, since several messages cannot be sent on the same stream by different clients. However, several messages can be sent on the same port, which means that ports can be embedded.
- With naive binary merging of streams, message sending delay is variable. With ports, message sending delay is constant.
- Objects based on streams require that the streams are closed when the clients stop using them. This is similar to decrementing a reference counter, and has similar problems, besides being unnecessarily explicit and low-level. A port is automatically closed when there are no more potential senders, thus notifying the object consuming messages.

11 Functions and Relations

Functions and relations are simple but powerful mathematical concepts. Many programming languages have been designed so that one of the available interpretations of a procedure definition should be a function or a relation. AKL has well-defined subsets that enjoy such interpretations, and provide the corresponding programming paradigms.

11.1 Functions

The functional style of programming is characterised by the determinate flow of control and by the non-cyclic flow of data. There is no don't care or don't know nondeterminism: a single result is computed; and agents do not communicate bi-directionally; an agent takes input from one agent and produces output to another agent. The latter point is weakened somewhat if the language has a non-strict semantics, in which case "tail-biting" techniques are possible.

Many of the AKL definitions are indeed written in the functional style. For example, the "append", "squares" and "fruits" definitions in the preceding sections are essentially functional, although the latter two were introduced as components in a process-oriented setting.

The basic relation between functional programs and AKL definitions is illustrated by an example, written in the non-strict, purely functional language Haskell. (The appropriate type declarations are supplied with the functional program for clarity.)

```
data (BinTree a) => (Leaf a) | (Node (BinTree a) (BinTree a))

flatten :: (BinTree a) -> [a]

flatten (Leaf x) l = x:l
flatten (Node x y) l = flatten x (flatten y l)
```

In AKL, a corresponding program is phrased as follows.

```
flatten(leaf(X), L, R) :-
      →    R = [X|L].
flatten(node(X, Y), L, R) :-
      →    flatten(Y, L, L1),
           flatten(X, L1, R).
```

The main differences are that an explicit argument has to be supplied for the output of the "function", and that nested function applications are un-nested, making the output of one the input of another.

AKL is not a higher-order language, and does not provide "definition variables", but does provide the same functionality (modulo currying) in a simple manner. The technique has been known in logic programming for a long time [Warren 1981]. A term representation is chosen for each definition in a program, and an agent apply is defined, which given such a term applies it to arguments and executes the corresponding definition.

One possible scheme for AKL is as follows. Let a term $p(n, t_1, \ldots, t_m)$ represent a definition $p/(n-m)$, which when applied to $n-m$ arguments $t_{m+1}, \ldots, t_n$ calls p/n with $p(t_1, \ldots, t_n)$.

To give an example relating to the above programs, the term flatten(3) corresponds to the function flatten, and the term flatten(3, Tree) to the function (flatten tree) (where Tree and tree are equivalent trees). A corresponding agent

```
apply(flatten(3), [X,Y,Z]) :-
      →    flatten(X, Y, Z).
apply(flatten(3,X), [Y,Z]) :-
      →    flatten(X, Y, Z).
apply(flatten(3,X,Y), [Z]) :-
      →    flatten(X, Y, Z).
apply(flatten(3,X,Y,Z), []) :-
      →    flatten(X, Y, Z).
```

is also defined. In practice, it is convenient to regard apply as being defined implicitly for all definitions in a program, which is also easily achieved in an implementation. This functionality may now be used as in functional programs as follows. We define an agent map/3, which maps a list to another list.

```
map(P, [], Ys) :-
      →    Ys = [].
map(P, [X|Xs], Ys0) :-
      →    Ys0 = [Y|Ys],
           apply(P, [X, Y]),
           map(P, Xs, Ys).
```

and may then call it with, e.g., map(append(3,[a]), [[b],[c]], Ys) and get the result Ys = [[a,b],[a,c]].

Although by no means necessary, expressions corresponding to lambda expressions can also be introduced. Let an expression

$$(\mathrm{X}_1, \ldots, \mathrm{X}_k)\backslash \mathrm{A}$$

where A is an AKL agent with free variables $\mathrm{Y}_1, \ldots, \mathrm{Y}_m$, stand for a term

$$\mathrm{p}((m+k), \mathrm{Y}_1, \ldots, \mathrm{Y}_m)$$

where $\mathrm{p}/(m+k)$ is a new agent defined as

$$\mathrm{p}(\mathrm{Y}_1, \ldots, \mathrm{Y}_m, \mathrm{X}_1, \ldots, \mathrm{X}_k) := \mathrm{A}.$$

We may now write, e.g., map((X,Y)\append(X, Z, Y), [[b],[c]], Ys) and get the result Ys = [[b|Z],[c|Z]]. Finally, the syntactic gap to the functional notation can be closed even further by introducing the syntax

$$\mathrm{P}(\mathrm{X}_1, \ldots, \mathrm{X}_k)$$

standing for

$$\mathrm{apply}(\mathrm{P}, [\mathrm{X}_1, \ldots, \mathrm{X}_k])$$

Obviously, the terms corresponding to functional closures may be given more efficient representations in an implementation.

11.2 Relations

The relational paradigm is known from logic programming as well as from from database query languages. Most prominent of logic programming languages is Prolog, which is entirely based on the relational paradigm. A large number of powerful programming techniques have been developed. Prolog and its derivatives are used for data and knowledge base applications, constraint satisfaction, and general symbolic processing. AKL supports Prolog-style programming.

Characteristic of the relational paradigm is the idea that programs interpreted as defining relations should be capable of answering queries involving these relations. Thus, if a parent relation is defined, the program should be able to produce all parents for given children and all children for given parents, enumerate all parents and corresponding children, and verify given parents and children. The following definition clearly satisfies this condition.

```
parent(sverker, adam).
parent(kia, adam).
parent(sverker, axel).
parent(kia, axel).
parent(jan_christer, sverker).
parent(hillevi, sverker).
```

Maybe less intuitive, but just as appealing, is the following: a simple parser of a fragment of the English language. The creation of a parse-tree is omitted.

```
s(S0, S) := np(S0, S1), vp(S1, S).

np(S0, S) := article(S0, S1), noun(S1, S).

article([a|S], S).
article([the|S], S).

noun([dog|S], S).
noun([cat|S], S).

vp(S0, S) := intransitive_verb(S0, S).

intransitive_verb([sleeps|S], S).
intransitive_verb([eats|S], S).
```

The two arguments of each atom represent a string of tokens to be parsed as the difference between the first and the second argument. The following is a sample execution.

```
s([a,dog,sleeps], S)
np([a,dog,sleeps], S2), vp(S2, S)
article([a,dog,sleeps], S1), noun(S1, S2), vp(S2, S)
noun([dog,sleeps], S2), vp(S2, S)
vp([sleeps], S)
intransitive_verb([sleeps], S)
S = []
```

The relation defined by s is

s([a, dog, sleeps|S], S)
s([a, dog, eats|S], S)
s([a, cat, sleeps|S], S)
s([a, cat, eats|S], S)
s([the, dog, sleeps|S], S)
s([the, dog, eats|S], S)
s([the, cat, sleeps|S], S)
s([the, cat, eats|S], S)

for all S, and will be generated as alternative results from

s(S_0, S)

The idea of a pair of arguments representing the difference between lists is important enough to warrant syntactic support in Prolog, the DCG syntax, which allows the above definitions to be rendered as follows.

```
s --> np, vp.

np --> article, noun.

article --> [a].
article --> [the].
```

and so on. The example is naive, since real examples would be unwieldy, but the state of the art is well advanced, and the literature on unification grammars based on the above simple idea is rich and flourishing.

12 Constraint Programming

Many interesting problems in computer science and neighbouring areas can be formulated as constraint satisfaction problems (CSPs). To these belong, for example, Boolean satisfiability, graph colouring, and a number of logical puzzles (a couple of which will be used as examples). Other, more application oriented, problems can usually be mapped to a standard problem, e.g., register allocation to graph colouring. Often, these problems are NP-complete; any known general algorithm will require exponential time in the worst case. Our task is to write programs that perform well in as many cases as possible.

A CSP can be defined in the following way. A *(finite) constraint satisfaction problem* is given by a sequence of variables $X_1, \ldots, X_n$; a corresponding sequence of (finite) domains of values $D_1, \ldots, D_n$; and a set of constraints $c(X_{i_1}, \ldots, X_{i_k})$. A *solution* is an assignment of values to the variables, from their corresponding domains, which satisfies all the constraints.

For our purposes, a constraint can be regarded as a logical formula, where satisfaction corresponds to the usual logical notion, but formalism will not be pressed here. Instead, AKL programs are used to describe CSPs, and their intuitive logical reading provides us with the corresponding constraints. Each agent is regarded as a (user-defined) constraint, and will be referred to as such. The agents are typically don't know nondeterministic, and the assignments for which the composition of these agents does not fail are the solutions of the CSP.

The example to be used in this section is the n-queens problem: how to place n queens on an n by n chess board in such a way that no queen threatens another. The problem is very well known, and no new algorithm will be presented. The novelty, compared to solutions in conventional languages, lies in the way the algorithm is expressed. The technique used is due to Saraswat [7].

Each square of the board is a variable V, which takes the value 0 (meaning that there is no queen on the square) or 1 (meaning that there is a queen on the square).

The basic constraint is that there may be at most one queen in each row, column, and diagonal. Given that n queens are to be placed on an n by n board, a derived constraint, which we will use, is that there must be exactly one queen in each row and column. Note that the exactly-one constraint can be decomposed into an at-least-one and an at-most-one constraint. We now proceed to define these constraints in terms of smaller components. The problem is not only to express the constraints, which is easy, but to express them in such a way that an appropriate level of propagation will occur, which will reduce the search space dramatically.

The at-most-one constraint can be expressed in terms of the following agent.

xcell(1, N, N).
xcell(0, _, _).

Note that this agent is determinate if the first argument is known, or if the last two arguments are known and different. For a sequence of squares V_1 to V_k, we can now express that at most one of these squares is 1 using the xcell agent as follows.

xcell(V_1, N, 1),
xcell(V_2, N, 2),
...,
xcell(V_k, N, k)

If more than one V_i is 1, the variable N will be bound to two different numbers, and the constraint will fail. Let us call this constraint at_most_one(V_1, ..., V_k), thus avoiding the overhead of having to write a program to create it.

An at_most_one constraint will clearly only have solutions where at most one square is given the value 1, but note also the following propagation effects. If one of the V_i is given the value 1, its associated xcell agent becomes determinate, and can therefore be reduced. When it is reduced, N is given the value i, and the other xcell agents become determinate, and can be reduced, giving their variables the value 0.

The at-least-one constraint can be expressed in terms of the following agent.

ycell(1, _, _).
ycell(0, S, S).

Note that this agent too is determinate if the first argument is known, or if the other two arguments are known and different. For a sequence of squares V_1 to V_k, we can express that at least one of these squares is 1 using the ycell agent as follows.

S_0 = begin,
ycell(V_1, S_0, S_1),
ycell(V_2, S_1, S_2),
...,
ycell(V_k, S_{k-1}, S_k),
S_k = end

If all the squares are 0, a chain of equality constraints, $S_0 = S_1$, $S_1 = S_2$, ..., will connect "begin" with "end" by equality constraints, and the constraint will fail. This constraint we call at_least_one(V_1, ..., V_k).

Again note the propagation effects. If a variable is given the value 0, then its associated ycell agent becomes determinate. When it is reduced, its second and third arguments are unified. If all variables but one are 0, the second argument of the remaining ycell agent will be "begin" and its third argument will be "end", and it will therefore be determinate. When it is reduced, its first argument will be given the value 1.

Thus, not only will these constraints avoid the undesirable cases, but they will also detect cases where information can be propagated. When no agent is determinate, and therefore no information can be propagated, alternative assignments for variables will be explored by trying alternatives for the xcell and ycell agents.

A program solving the n-queens problem can now be expressed as follows.

- For each column, row, and diagonal, consisting of a sequence of variables V_1, ..., V_k, the constraint at_most_one(V_1, ..., V_k) has to be satisfied.
- For each column and row, consisting of a sequence of variables V_1, ..., V_n, the constraint at_least_one(V_1, ..., V_n) has to be satisfied.
- The composition of these constraints is the program.

Note that when information is propagated, this will affect other agents, making them determinate. This will often lead to new propagation. One such case is illustrated below.

1	0	0	0
0	0	V_{23}	V_{24}
0	V_{32}	0	V_{34}
0	V_{42}	V_{43}	0

The above grid represents the board, and in each square is written the variable representing it, or its value if it has one. We will now trace the steps leading to the above state. Initially, all variables are unconstrained, and all the constraints have been created. Let us now assume that the topmost leftmost variable (V_{11}) is given the value 1. It appears in the row V_{11} to V_{14}, in the column V_{11} to V_{41}, and in the diagonal V_{11} to V_{44}. Each of these is governed by an at_most_one constraint. By giving one variable the value 1, the others will be assigned the value 0 by propagation.

A second case of propagation is the following, where V_{12} and V_{24} are assumed to contain queens, and propagation of the above kind has taken place.

0	1	0	0
0	0	0	1
V_{31}	0	0	0
V_{41}	0	V_{43}	0

Here we examine the propagation that this state will lead to. Notice that in row 3, all variables but V31 have been given the value 0. This triggers the at_least_one constraint governing this row, giving the last variable the value 1, which in turn

gives the variables in the same row, column, or diagonal (only V_{41}) the value 0. Finally, V_{43} is given the value 1 by reasoning as above.

The above program is reasonably good. The programs usually written for constraint logic programming languages with finite domain constraints do not exploit the fact that both rows and columns should contain exactly one queen (e.g., [10]). A very good solution can be obtained if the xcell and ycell agents are ordered so that those governing variables closer to the centre of the board come before those governing variables further out. If at some step alternatives have to be tried for an agent, values will be guessed for variables at the centre first. This happens to be a good heuristic for the n-queens problem, even better than the so called first-fail principle, which is usually employed.

13 Integration

So far, the different paradigms have been presented one at a time, and it is quite possibly by no means apparent in what relation they stand to each other. In particular the relational and the constraint satisfaction paradigms have no apparent connection to the process paradigm. Here, this apparent dichotomy will be bridged, by showing how a process-oriented application based on the solver for the *n*-queens problem could be structured.

The basic techniques for interaction with the environment (e.g., files and the user) are shown first, and then a program structure is introduced which is somewhat inspired by the Smalltalk Model-View-Controller paradigm.

13.1 Interoperability

The idea underlying interoperability is that an AKL agent sees itself as living in a world of AKL agents. The user, files, other programs, all are viewed as AKL agents. If they have a state, e.g., file contents, they are closer to objects, such as those discussed above. It is up to the AKL implementation to provide this view, which is inherited from the concurrent logic programming languages.

A program takes as parameter a port to the "operating system" (OS) agent, which provides further access to the functionality and resources it controls. An interface to foreign procedures adds glue code that provides the necessary synchronisation, and views of mutable data structures as ports to agents. The examples use imaginary, although realistic, primitives, as in the following.

```
main(P) :=
        send(create_window(W, [xwidth=100, xheight=100]), P),
        send(draw_text(10, 10, 'Hello, world!'), W).
```

Here it is assumed that the agent main is supplied with the "operating system" port P when called. The OS provides window creation, an operation that returns a port to the window agent, which provides text drawing, and so on.

For some kinds of interoperability, a consistent view of don't know nondeterminism can be implemented. For example, a sub-program without internal state, such as a numerical library written in C, does not mind if its agents are copied during the course of a computation. For particular purposes, it is even possible to copy windows and similar "internal" objects. But the real world does not support don't

know nondeterminism. It would hardly be possible to copy an agent that models the actual physical file-system; nor would it be possible to copy an agent that models communication with another computer.

The only solution is to regard this kind of incompleteness as acceptable, and either let attempts to copy such unwieldy agents induce a run-time error, or give statements a "type" which is checked at compile-time, and which shows whether a statement can possibly employ don't know nondeterminism.

13.2 Encapsulation

To avoid unwanted interaction between don't know nondeterministic and process-oriented parts of a program, the nondeterministic part can be encapsulated in a statement that hides nondeterminism. Nondeterminism is encapsulated in the guard of a conditional or committed choice and in bagof. When encapsulated in a guard, a nondeterministic computation will eventually be pruned. In a conditional choice, the first solution is chosen. In a committed choice, any solution may be chosen. When encapsulated in bagof, all solutions will be collected in a list.

More flexible forms of encapsulation can be based on the notion of engines. An engine is conceptually an AKL interpreter. It is situated in a server process. A client may ask the engine to execute programs, and, depending on the form of engine, it may interact with the engine in almost any way conceivable, inspecting and controlling the resulting computation. A full treatment of engines for AKL is future work.

13.3 Model-View-Controller

The Model-View-Controller (MVC) paradigm for assigning different responsibilities to the components of an object-oriented program is easily realised in AKL, as in any object- or process-oriented language. In AKL it also localises and encapsulates don't know nondeterminism in the relevant part of the program, which is usually the model.

In the next section, MVC will be applied to the structuring of an n-queens application, using imaginary OS primitives.

13.4 An N-Queens Application

Assume the existence of a don't know nondeterministic n-queens agent

n_queens(N, Q) := ....

which returns different assignments Q to the squares of an N by N chess board. It is easily defined by adding code for creation of constraints for different length sequences, and code for creating sequences of variables corresponding to rows, columns, and diagonals on the chess board. No space will be wasted on this trivial task here. We proceed to the MVC structure with which to support the application.

```
main(P) :=
        initialise(P, W, E),
        view(V, W),
        controller(E, M, S, V),
        model(M, S).
```

The initialise agent creates a window accepting requests on stream W and delivering events on stream E. The view agent presents whatever it is told to by the controller on the window using stream W. The model delivers solutions on the stream S to the n-queens problems submitted on stream M. The controller is driven by the events coming in on E. It submits problems to the model on stream M and receives solutions on stream S. It then sends solutions to the view agent on V for displaying.

Let us here ignore the implementation of the initialise, view, and controller agents. The interesting part is how the don't know nondeterminism is encapsulated in the model agent. We assume that we are satisfied with being able to get either one or all solutions from the particular instance of the n-queens problem, or getting the reply that there are no solutions (for N = 2 or N = 3).

```
model([], S) :-
      →    S = [].
model([all(N)|M], S) :-
      →    bagof(Q, n_queens(N, Q), Sols),
           S = [all(Sols)|S1],
           model(M, S1).
model([one(N)|M], S) :-
      →    ( Q :       n_queens(N, Q)
                →      S = [one(Q)|S1]
           ;           S = [none|S1] ),
           model(M, S1).
```

As described above, don't know nondeterminism within bagof and choice statements is not propagated further. The MVC part of the program can be kept entirely free of nondeterminism.

14 Current and Future Work

Current and planned topics at SICS include efficient sequential and parallel implementations parametrised with user-definable constraint systems (in C), implementations of various constraint systems, extensions of the basic framework, such as engines for meta-level programming, program analysis and program transformation, inter-operability with conventional languages and operating systems, and investigation of formal properties.

An experimental AKL programming system is available from SICS for research and educational purposes.

Acknowledgements

The authors wish to thank the other members of the Concurrent Constraint Programming group at SICS for their contributions to this work. Discussions with Vijay Saraswat and David H. D. Warren during the design phase were very valuable.

References

[1] Andrew Davison. *POLKA: A Parlog Object-Oriented Language.* PhD thesis, Department of Computing, Imperial College, London, May 1989.

[2] Yaron Goldberg and Ehud Shapiro. Logic programs with inheritance. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*. Omsha Ltd, June 1992.

[3] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium*. MIT Press, 1991.

[4] Sverker Janson, Seif Haridi, and Johan Montelius. *Research Directions in Concurrent Object-Oriented Programming*, chapter Ports for Objects in Concurrent Logic Programs. MIT Press, 1993.

[5] Sverker Janson and Johan Montelius. The design of the AKL/PS 0.0 prototype implementation of the Andorra Kernel Language. ESPRIT deliverable, EP 2471 (PEPMA), Swedish Institute of Computer Science, 1992.

[6] Kenneth M. Kahn, Eric Dean Tribble, Mark S. Miller, and Daniel G. Bobrow. Vulcan: Logical concurrent objects. In P. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[7] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. MIT Press, 1993.

[8] Ehud Shapiro, editor. *Concurrent Prolog: Collected Papers*. MIT Press, 1987.

[9] Ehud Shapiro and Akikazu Takeuchi. Object-oriented programming in Concurrent Prolog. *Journal of New Generation Computing*, 1(1):25–49, 1983.

[10] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[11] K. Yoshida and T. Chikayama. A'UM—a stream-based concurrent object-oriented language. In *Proceedings of FGCS'88*, ICOT, Tokyo, 1988.