```
spawn_monitors([], P, B) :-
    → true.
spawn_monitors([V | L], P, B) :-
    → monitor(V, P, B),
      spawn_monitors(L, P, B).

monitor(V, P, B) :-
      data(V)
    | send(V, P).
monitor(V, P, B) :-
      data(B)
    | V = B,
      send(V, P).

controller(S, 0, K, B) :-
    | B = 0,
      turned(S, K, 0).
controller(S, K, K, B) :-
    | B = 1,
      turned(S, K, 1).
controller([1 | S], N, K, B) :-
      N > 0
    | controller(S, N-1, K-1, B).
controller([1 | S], N, K, B) :-
      K > N
    | controller(S, N, K-1, B).

turned(0, S, V) :-
    → S = [].
turned(N, [V | S], V):-
      N > 0
    → turned(N-1, S, V).
```

This program solves the problem illustrated in Figure 8.3 deterministically.

### *8.3.1 The Cardinality Operator*

An elegant generalisation of the monitor-controller technique is the implementation in AKL of the *cardinality operator* [Van Hentenryck and Deville 1991]. The computation rules given are straight-forwardly realised in AKL as the different clauses of a controller.

The cardinality operator is an agent #(L, U, Cs), where L and U are natural numbers and Cs is a given list of (representations of) constraints. It succeeds if the number of constraints that are true is in the range L to U, the lower and upper bounds. Furthermore, its propagation effect is that if it at some point is known that the remaining constraints must be all false, or all true, this knowledge is published immediately.

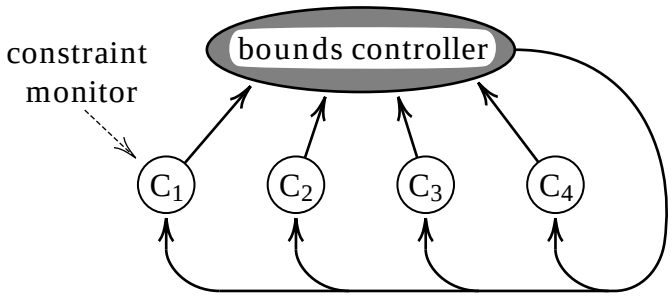


Figure 8.6. Monitor-controller for the cardinality operator

Clearly, the at-most-one, at-least-one, and exactly-one constraints are subsumed by this general construct, which is expressed in AKL as follows.

```
#(L, U, Cs) :=
        open_port(P, S),
        constraints(Cs, N, P, F),
        bounds(L, U, N, S, F).
```

The *trivial satisfaction* rule:

```
bounds(L, U, N, S, F) :-
        L =< 0, N =< U
    |   F = any.
```

The *positive satisfaction rule*:

```
bounds(L, U, N, S, F) :-
        L =< U, L = N
    |   F = true,
        allequal(N, true, S).
```

The *negative satisfaction rule*:

```
bounds(L, U, N, S, F) :-
        L =< U, U = 0
    |   F = false,
        allequal(N, false, S).
```

The *positive reduction rule*:

```
bounds(L, U, N, [true | S], F) :-
        ( 0 < L, L < N, L =< U
        ; 0 < U, U < N, L =< 0 )
    |   bounds(L-1, U-1, N-1, S, F).
```

The *negative reduction rule*:

```
bounds(L, U, N, [false | S], F) :-
        ( 0 < L, L < N, L =< U
        ; 0 < U, U < N, L =< 0 )
    |   bounds(L, U, N-1, S, F).
```

The constraints agent spawn monitors for the constraints.

```
constraints([], 0, _, F) :-
    → true.
constraints([C | Cs], N, P, F) :-
    → constraint(C, P, F),
      constraints(Cs, N1, P, F),
      N = N1+1.
```

For simplicity, we assume that the representation of constraints is callable. We also use general negation to express the published negative constraint. This could be replaced by some general means of interpreting the representation.

```
constraint(C, P, F) :-
        call(C)
    |   send(true, P).
constraint(C, P, F) :-
        not(call(C))
    |   send(false, P).
constraint(C, P, true) :-
    |   call(C).
constraint(C, P, false) :-
    |   not(call(C)).
constraint(C, P, any) :-
    |   true.
```

The allequal agent verifies that all incoming messages have the same value as that broadcast to the remaining monitors.

```
allequal(0, X, S) :-
    → S = [].
allequal(N, X, S) :-
        N > 0
    → allequal(N-1, X, S1),
      S = [X | S1].
```

### *8.3.2 Local Search*

Closely related to the notion of independent and-parallelism achieved by local execution in guards is a technique for writing search programs in AKL.

In this example, due to D.H.D. Warren, we illustrate both the incremental growth of or-trees in AKL, and how the notion of local execution in guards makes it possible to write elegant search programs. It is a program that finds common sub-lists of two lists. We start by giving a Prolog/CLP program for this task.

```
sublist([], Y).
sublist([E | X], [E | Z]) :- sublist(X, Z).
sublist(X, [E | Y]) :- X = [_ | _], sublist(X, Y).

?- sublist(L, [c,a,t,s]), sublist(L, [l,a,s,t]).
```

This program will repeat the execution of the second goal for each solution of the first goal. One could imagine evaluating the first or the second goal in advance, and combining its results with the solutions of the other. But this is not a good solution. The goals have a large number of solutions, many of which are completely irrelevant. Clearly, there is a trade-off between the drawback of repeating work and the drawback of wasting work.

However, it is reasonably safe to execute each goal locally until the first point at which the execution for this goal could fail (in an "interesting" way). This happens when the first element of a sublist is generated. To achieve this effect, the above definition is transformed into the following.

```
sublist([], Y).
sublist([E | X], Y) :- suffix([E | Z], Y) ? sublist(X, Z).

suffix(X, X).
suffix(X, [E | Y]) :- suffix(X, Y).
```

Let us start with the initial goal

```
sublist(L, [c,a,t,s]), sublist(L, [l,a,s,t])
```

The sublist goals are first unfolded (into the implicit nondeterminate choice statements).

```
( L=[] ? true
; E, L1, Z : L=[E | L1], suffix([E | Z], [c, a, t, s]) ? sublist(L1, Z) ),
( L=[] ? true
; E, L1, Z : L=[E | L1], suffix([E | Z], [l, a, s, t]) ? sublist(L1, Z) )
```

The guards (preceding the wait operators "?") will now execute locally, finding all solutions. Without getting into details, it should be fairly intuitive that, after a while, the following configuration is reached.

```
( L = [] ? true
; L1 : L = [c | L1] ? sublist(L1, [a, t, s])
; L1 : L = [a | L1] ? sublist(L1, [t, s])
; L1 : L = [t | L1] ? sublist(L1, [s])
; L1 : L = [s | L1] ? sublist(L1, []) ),
( L = [] ? true
; L1 : L = [l | L1] ? sublist(L1, [a, s, t])
; L1 : L = [a | L1] ? sublist(L1, [s, t])
; L1 : L = [s | L1] ? sublist(L1, [t])
; L1 : L = [t | L1] ? sublist(L1, []) )
```

The solutions that have been found for the suffix/2 goal give rise to different alternative guarded goals in the choice statement. At this stage, these alternatives have to be tried. Further computation will eventually lead to a "Cartesian product" of the alternatives in the two choice statement. When the combinations that fail have been removed, the combined alternatives are as follows.

$L = []$

$L = [a \mid L_1]$, sublist($L_1$, [t, s]), sublist($L_1$, [s, t])

$L = [t \mid L_1]$, sublist($L_1$, [s]), sublist($L_1$, [])

$L = [s \mid L_1]$, sublist($L_1$, []), sublist($L_1$, [t])

We have one complete solution L = []. Execution will now continue within each of the remaining alternatives, but we will stop here, hopefully having demonstrated the main point.

## 8.4 AKL VS. THE CC FRAMEWORK

The concept of *concurrent constraint programming*, due to seminal work by Saraswat [1989; Saraswat and Rinard 1990; Saraswat, Rinard, and Panangaden 1991], evolved from earlier work in constraint logic programming [Jaffar and Lassez 1987] and in concurrent logic programming [Maher 1987; Saraswat 1987]. One source of inspiration was the logical view of committed-choice language provided by Maher [1987], where the entailment-based view of synchronisation was introduced.

For his analysis of the concurrent constraint programming paradigm, Saraswat developed a family of languages called the cc framework, which are in many ways the immediate ancestors of AKL. Saraswat has pursued the study of these languages in new directions, for example basing them on Linear Logic [Girard 1987], but the comparison made here focuses on his original cc framework [Saraswat 1989].

The core of the cc family is a language called cc(↓,→,⇒), providing blocking Ask "↓", Atomic Tell "★", indeterminate (don't care) prefixing "→", and nondeterministic (don't know) prefixing "⇒". The cc family is open-ended, in that other combinators can be conceived and included, such as Eventual Tell and groups, which are of interest for the following comparison with AKL.

Many of the ideas in the cc languages recur in AKL, although rearranged in a smaller set of higher level primitives with improved control and synchronisation. Saraswat proposed a control for cc languages which is based on the Basic Andorra Model (Section 8.1.8). AKL generalises this form of control by also taking into account the interactions between constraint stores in the hierarchy formed by the nesting of choice and bagof statements. AKL control would also be applicable to cc languages.

In the following sections the most important combinators of the cc family are compared with corresponding constructs in AKL, and finally a mapping from a restriction of the language cc(↓,*,→,⇒) to AKL is shown.

### *8.4.1 Basic Operations*

Some basic operations correspond to implicit operations on constraints in AKL.

The *Ask* ("↓") operation corresponds closely to the ask operation implicitly employed by AKL for various purposes. For example, the behaviour of a cc agent

$(c\downarrow \rightarrow A)$ may be translated into AKL as $(c \rightarrow A^*)$, where $^*$ is a mapping from cc agents to AKL statements. The difference between this and the cc operation is that asking is done with respect to local variables, which are existentially quantified. An AKL agent $(x : c(x) \mid A(x))$ asks $\exists x\, c(x)$, and may then use $x$ in A. This behaviour can be achieved with a cc agent $((\exists x\, c(x))\downarrow \rightarrow (c(x)* \rightarrow A(x)))$, but at the cost of repetition, which, apart from being verbose, could correspond to similar repetitious behaviour in the manipulation of constraints in an implementation.

The *Eventual Tell* ("$*$") operation is similar to the tell operation implicitly employed by AKL. The difference is that the told constraints are not buffered in AKL; instead, the inconsistency of the resulting constraint store may be detected at a later time. AKL uses telling on constraint atoms and for promotion of constraints computed in guards.

The *Atomic Tell* ("$\star$") operation has no corresponding operation in AKL. Similar effects can be achieved with the *cut* operation (proposed as an extension of AKL in Section 4.8). This operation requires that the state is stable. There is also an ordering of agents in AKL which will allow no competition in this situation: the leftmost will be chosen. Some uses of atomic tell can be emulated, but with less concurrency, and hence less potential parallelism. The most important reason for abstaining from atomic tell is the difficulty of efficient parallel implementation. The atomicity requirement means that an arbitrary number of variables have to be locked. With the stability condition it is guaranteed that no other agents can interfere, but at the cost of less concurrency.

Saraswat also proposes so called *atomic groups*: the "guard" and the "wfguard" constructs. They are basic operations, and may thus be used as the prefix of a prefixing operation. Their definition is not compatible with the rest of the framework (without modification), but intuitively they provide user-defined atomic transactions. AKL guard execution is always atomic in this sense, due to the separation of constraint stores. The implementation scheme proposed for atomic groups involves copying the entire constraint store, and is hardly realistic for the intended uses.

In addition the cc framework has the Global Ask, Ask-and-Instantiate, Ask-and-fix, Inform, and Check basic operations, which have no counterparts in AKL, in which the constraint operations are monotone and do not introduce nondeterminism.

### *8.4.2 Prefixing Combinators*

Prefixing combinators are similar to the guard operators of AKL. The comparisons are done with respect to the choice statements corresponding to these guard operators.

*Nondeterministic prefixing* ("$\Rightarrow$") is similar to nondeterminate choice in AKL. Any basic operation may be used as prefix. When the operation in the prefix succeeds, a don't know nondeterministic split may be performed, with no additional synchronisation requirements. This is to be compared with AKL which

uses a form of telling in the guard of the nondeterminate choice statement, and which may only perform don't know nondeterministic actions in stable states, when nothing can be done to avoid it. Nondeterministic prefixing is "parallel" in that the different alternative computations may be explored concurrently. Saraswat also proposes a sequential variety, which explores one alternative at a time, the others being blocked until it terminates. This does not correspond to anything in AKL; the reasons for including it or avoiding it are similar to those for sequential composition.

*Indeterminate prefixing* ("→") is similar to committed choice in AKL. Again, any basic operation may be used as prefix. The effect achieved when Ask is used was discussed above. The effect achieved when Atomic Tell is used was also discussed above. The effect achieved when Eventual Tell is used is not likely to be useful.

### *8.4.3 Coarse-Grained Combinators*

Saraswat recognises the need to encapsulate nondeterministic computations, and proposes a few coarse-grained interleaved combinators for this purpose.

The *single-solution group* ("oneg") is similar to a guard of a committed choice statement in that it selects a single solution of an otherwise encapsulated computation. The differences are that there is no entailment requirement, nor a "continuation" in the form of a body, which performs a particular action with respect to the chosen solution. The lack of synchronisation makes the choice of solution more arbitrary than necessary.

The *all-solution group* ("allg") is similar to a guard of a nondeterminate choice statement in that all solutions of an otherwise encapsulated computation will be tried nondeterministically. Again, the lack of synchronisation makes the choice of solutions to be tried quite arbitrary.

The *reconciliation* combinator ("⇑") provides the ability to take the "Cartesian product" of two deadlocked groups (of the same kind). A related effect can be achieved in AKL by explicit decomposition of problems into what should be executed locally, and what should be a residue to be combined (Section 8.6).

The *synthesise* combinator ("synthesize") communicates nothing until all solutions of an encapsulated computation have been found, whereupon it combines these into a single disjunctive constraint, which is told to the store. No corresponding operation is available in AKL (although a number of candidates have been discussed).

The definition of the above combinators is not compatible with the rest of the framework (without modification). The resulting computation model would have a flavour reminiscent of AKL, in that it provides a hierarchy of constraint stores.

### *8.4.4 Essential Combining Forms*

And-parallel composition ("||") corresponds to composition in AKL.

Or-selection ("□") corresponds to, if anything, the ";" operator separating the guarded statements in choice statements. In AKL the guarded statements in a choice statement must all be of the same kind, i.e., the behaviour is associated with the choice statement. Although the cc framework allows combinations of different prefixing combinators, this ability is not much used.

Existential quantification ("^") corresponds to hiding in AKL.

### *8.4.5 cc Programs in AKL*

We will map a subset of cc(↓,∗,→,⇒), with Eventual Tell instead of the usual Atomic Tell, to corresponding AKL programs. We assume the algebraic syntax of cc programs in the sense that or-selection separates the clauses in a definition, otherwise the standard syntax is used. We further observe that, with Eventual Tell, $(c\downarrow : c* \rightarrow A)$ is equivalent to $(c\downarrow \rightarrow (c* \rightarrow A))$.

Thus, we assume definitions in the following form.

$$A ::= \text{stop} \mid \text{fail} \mid g \mid A \parallel A$$

$$B ::= c* \mid c\downarrow$$

$$D ::= X\,\hat{}\,D \mid B \Rightarrow A \mid B \rightarrow A \mid D \,\square\, D$$

$$K ::= g :: D$$

We further add the restriction that all clauses in a definition must have the same prefixing combinator. The mapping $^*$ assigns a corresponding AKL program to a given cc(↓,∗,→,⇒) program with the given restriction. (Please excuse the overloading of the "⇒" arrow.)

$$(\text{stop} \mid \text{fail} \mid g)^* \Rightarrow (\text{true} \mid \text{fail} \mid g)$$

$$(A \parallel A)^* \Rightarrow (A^*, A^*)$$

$$(c\downarrow \rightarrow A)^* \Rightarrow c \mid A^*$$

$$(c* \rightarrow A)^* \Rightarrow \text{true} \mid c, A^*$$

$$(c\downarrow \Rightarrow A)^* \Rightarrow (c \mid \text{true})\,?\,A^*$$

$$(c* \Rightarrow A)^* \Rightarrow c\,?\,A^*$$

$$(X\,\hat{}\,D)^* \Rightarrow (X : D^*)$$

$$(D_1 \,\square\, \cdots \,\square\, D_n)^* \Rightarrow (D_1{}^* ; \cdots ; D_n{}^*)$$

$$(g :: D)^* \Rightarrow g := D^*$$

For example, the cc program

```
max(X, Y, Z) ::
    ( X ≤ Y↓ ⇒ equal(Y, Z)
    □ Y ≤ X↓ ⇒ equal(X, Z) ).
```

```
equal(Y, Z) ::
    (Y = Z)* ⇒ stop.
```

translates into AKL as

```
max(X, Y, Z) :=
    ( (X ≤ Y | true) ? equal(Y, Z)
    ; (Y ≤ X | true) ? equal(X, Z) ).

equal(Y, Z) :=
    ( (Y = Z) ? true ).
```

The reason for asking in cc is often synchronisation. In AKL, the program would be written as

```
max(X, Y, Z) :=
    ( X ≤ Y % Y = Z
    ; Y ≤ X % X = Z ).
```

where '%' is either → (normally), | (hardly), or ? (possibly), depending on the desired behaviour. In the latter case synchronisation would be given by the nondeterminacy of the choice.

### *8.4.6 Conclusion*

AKL extends the expressive power of the cc family by improving the functionality which was aimed at by the coarse-grained combinators. Improved control, given by stability, and aggregates such as bagof, enable the use in AKL of don't know nondeterminism in otherwise reactive programs.

Saraswat has reconstructed the cc family in terms of Linear Logic [Girard 1987], yielding the Lcc and HLcc family of *linear* and *higher-order linear* concurrent constraint languages. This clearly adds expressiveness, but possibly at the price of sacrificing simplicity. The path chosen for AKL is to augment the language where a need has been perceived, such as adding *ports* for process communication, which rhymes well with the constraint-based communication scheme.

## 8.5 AKL VS. OZ

Oz is a deep-guard higher-order concurrent constraint programming language based on the constraint system of records [Smolka, Henz, and Würtz 1993; Smolka 1994; Smolka et al. 1994]. Oz is a descendant of AKL, and is its closest relative, but goes beyond it by providing new functionality such as *higher-orderness* and *processes* (threads). The standard formulation of Oz does not, however, provide don't know nondeterminism, although such extensions are being considered, and will surely become part of the language [Schulte and Smolka 1994].

Kernel Oz, the essential sublanguage to which other constructs can be reduced, has the following syntax.

| | | | |
|---|---|---|---|
| E | ::= | **false** \| **true** \| $x = s$ | constraints |
| | \| | $x = l(y_1 \ldots y_n)$ | tuple construction |
| | \| | $x = l(l_1 : y_1 \ldots l_n : y_n)$ | record construction |
| | \| | **proc** $\{ x \ \overline{y} \}$ $E$ **end** | procedure definition |
| | \| | $\{ x \ \overline{y} \}$ | procedure application |
| | \| | $E_1 \ E_2$ | concurrent composition |
| | \| | **local** $\overline{x}$ **in** $E$ **end** | variable declaration |
| | \| | **if** $C_1$ [] … [] $C_n$ **else** $E$ **fi** | conditional |
| | \| | **or** $C_1$ [] … [] $C_n$ **ro** | disjunction |
| | \| | **process** $E$ **end** | process creation |
| $C$ | ::= | $\overline{x}$ **in** $E_1$ **then** $E_2$ | clause |
| $x, y, z$ | ::= | ⟨variable⟩ | |
| $l$ | ::= | $x$ \| ⟨atom⟩ | |
| $s$ | ::= | $l$ \| ⟨number⟩ | |
| $\overline{x}, \ \overline{y}$ | ::= | ⟨possibly empty sequence of variable⟩ | |

Kernel procedures provide for constraint communication and additional constraint operations. The operational semantics is given by the Oz calculus, which is a rewrite system modulo structural congruence [Smolka 1994]. The Oz user language provides a plethora of syntax for different modes of expression in an ALGOL-like style.

The Oz conditional is a combination of AKL committed and conditional choice. It may commit to any of the clauses, but if all fail, the else-branch is chosen. It can be modelled in AKL in a manner analogous to the treatment of the sequential and parallel clause composition of PARLOG.

The Oz disjunction is similar to AKL nondeterminate choice, but will promote not only a clause with a solved guard, but any single remaining clause. This exact behaviour cannot be modelled in AKL. In the standard formulation, Oz does not have choice splitting, but in the extension for encapsulated search, disjunction plays the rôle of nondeterminate choice for splitting operations. This extension does not, however, make don't know nondeterminism orthogonal as in AKL. Splitting does not distribute over guards, but is only performed in a special *solve* statement. This makes don't know nondeterminism unsuitable for uses such as simple tests in guards, and the language loses some logic programming flavour. The intended use of the solve statement is that different branches of a search tree are explored by concurrent composition (“conjunction”). This makes it difficult to model single solution search without restricting the potential for parallelism, a problem shared with the committed-choice languages [Gregory 1993]. It is, however, also possible to explore the different branches in the guards of a conditional, in which case this problem does not arise. Finally, it is not clear how to achieve independent and-parallelism for search problems. The above comments pertain to a proposed extension, and may not be true of the final version [Schulte and Smolka 1994].