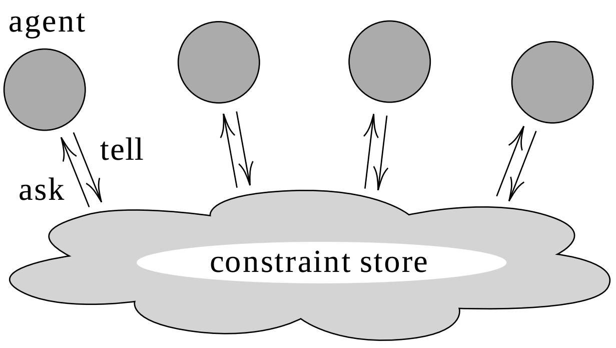

Figure 2.1. Agents interacting with a constraint store

The range of constraints that may be used in a program is defined by the current *constraint theory*, 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 theories as such are not investigated in this dissertation. For the purpose of this dissertation, we will use a simple constraint theory with a few obvious constraints, which is essentially that of Prolog and GHC with arithmetic added.

Thus, *constraints* in AKL will be formulas of the form

⟨expression⟩ = ⟨expression⟩

⟨expression⟩ ≠ ⟨expression⟩

⟨expression⟩ < ⟨expression⟩

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, …

or *numbers*, e.g.,

1, 3.1415, -42, …

or *arithmetic expressions*, e.g.,

1 + X, -Y, X * Y, …

or *constants*, e.g.,

a, b, c, …

or *constructor expressions* of the form

⟨name⟩(⟨expression⟩, …, ⟨expression⟩)

where ⟨name⟩ is an alpha-numeric symbol with a lower case initial letter, e.g.,

s(s(0)), tree(X, L, R), …

There is also the constant [], which denotes the *empty list*, and the *list constructor* [⟨expression⟩ | ⟨expression⟩]. 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.

Clearly, asking and telling numerical constraints can be computationally demanding (even undecidable). In this dissertation, only naïve capabilities are assumed, such as deriving $X = 3$ from $X = Y + 1$ and $Y = 2$ by simple propagation of values.

2.3 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* of the form

⟨name⟩(X_1, …, X_n)

is a defined agent. In a program atom, ⟨name⟩ is an alpha-numeric symbol and n is the *arity* of the atom. The variables X_1, …, 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 ⟨name⟩ / 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

⟨name⟩(X_1, …, X_n) := ⟨statement⟩.

The variables X_1, …, 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 ⟨name⟩ / n atom, e.g.,

plus(X, Y, Z) := Z = X + Y.

is a definition of plus / 3.

A *composition* statement of the form

⟨statement⟩, …, ⟨statement⟩

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

A *conditional choice* statement of the form

$(\langle\text{statement}\rangle \rightarrow \langle\text{statement}\rangle ; \langle\text{statement}\rangle)$

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 in Section 2.5 and arbitrary statements as conditions in Section 2.6.

A *hiding* statement of the form

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

introduces variables with local scope. The behaviour of a hiding statement is to replace itself with its component statement, in which the variables $X_1, \ldots, X_n$ have been replaced by new variables.

Let us at this point establish some syntactic conventions.

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

  X : p, q, r

  means

  X : (p, q, r)

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

  (X : p), q, 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 governed 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.

These syntactic conventions are always assumed, unless explicitly suppressed, for example in formal manipulations of the language.

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

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

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. For example,

```
append([1 | W], Y, Z)
```

may be rewritten to

```
( [1 | W] = [] → Z = Y
; [1 | W] = [E | X₁], append(X₁, Y, Z₁), Z = [E | Z₁] )
```

by unfolding the append atom, then to

```
append(W, Y, Z₁), Z = [1 | Z₁]
```

by executing the choice statement, and substituting values for variables according to the told equality constraint, then to

```
( W = [] → Z₁ = Y
; W = [E | X₁], append(X₁, Y, Z₂), Z₁ = [E | Z₂] ),
Z = [1 | Z₁]
```

by unfolding the append atom. At this point, the computation suspends. If another agent tells a constraint on W, e.g., W = [2, 3], the computation may be resumed and the final value Z = [1, 2, 3] can be computed. Thanks to the implicit ask synchronisation in conditional choice, the final result of a call to append does not depend on the order in which different agents are processed.

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.

```
list(N, L) :=
    ( N = 0 → L = []
    ; L = [N | L₁], list(N - 1, L₁) ).

sum(L, N) :=
    ( L = [] → N = 0
    ; L = [M | L₁], sum(L₁, N₁), N = N₁ + 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,

list(3, L), sum(L, N)

is rewritten to

list(2, L_1), 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

list(1, L_2), sum([2 | L_2], N_1), N = 3 + N_1

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

list(0, L_3), sum([1 | L_3], N_2), N = 5 + N_2

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).

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.

2.4 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

```
( ⟨statement⟩ | ⟨statement⟩
; ···
; ⟨statement⟩ | ⟨statement⟩ )
```

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

```
⟨statement⟩ | ⟨statement⟩
```

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

$X_1, \ldots, X_n$: ⟨statement⟩ | ⟨statement⟩

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, it and its corresponding body replace 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, "∃Y"). For example, in

```
X = f(a), ( Y : X = f(Y) | q(Y) )
```

the asked constraint is ∃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.

```
merge(X, Y, Z) :=
    ( X = [] | Z = Y
    ; Y = [] | Z = X
    ; E, X1 : X = [E | X1] | Z = [E | Z1], merge(X1, Y, Z1)
    ; E, Y1 : Y = [E | Y1] | Z = [E | Z1], merge(X, Y1, Z1) ).
```

A merge agent can react as soon as either X or Y is given a value. In the last two guarded statements, 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.

2.5 DON'T KNOW NONDETERMINISM

Many problems, especially frequent in the field of Artificial 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* statement.

```
        ( ⟨statement⟩ ? ⟨statement⟩
        ; …
        ; ⟨statement⟩ ? ⟨statement⟩ )
```

The symbol “?” is read *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 remaining guard and its corresponding body replace 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 may be 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 also 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 below.

2.6 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. Such guards are often referred to as *flat*. This will now be generalised to *deep* guards, which contain arbitrary statements.

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

```
( ⟨statement⟩ → ⟨statement⟩
; …
; ⟨statement⟩ → ⟨statement⟩ )
```

The symbol "→" is read *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

```
( ⟨statement⟩ → ⟨statement⟩
; true → ⟨statement⟩ )
```

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

```
( ⟨statement⟩ → ⟨statement⟩
; …
; true → ⟨statement⟩ )
```

may always be abbreviated to

```
    ( ⟨statement⟩ → ⟨statement⟩
    ; …
    ; ⟨statement⟩ )
```

For the last time we make the simplifying assumption that the guards are all constraints. The conditional choice statement asks the first guard. If it is entailed, it and its corresponding body replace 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).

```
  ( append(X, Y, Z) → p(Z)
  ; true → q(X, Y) )
```

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

```
  ( Z = [1,2,3] → p(Z)
  ; true → q(X, Y) )
```

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, known as their *environment*. 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.

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