8.2 AKL VS. COMMITTED-CHOICE LANGUAGES

The field of concurrent logic programming was initiated by Clark and Gregory [1981] with the Relational Language, and attracted considerable interest after strong promotion of Concurrent Prolog by Shapiro [1983]. Later developments include PARLOG [Clark and Gregory 1986; Gregory 1987], GHC [Ueda 1985], KL1 [Ueda and Chikayama 1990], and Strand [Foster and Taylor 1989; 1990].

With Concurrent Prolog, Shapiro first demonstrated the expressiveness of concurrent logic programming, and the process-oriented programming techniques developed have become standard. However, the synchronisation mechanism of Concurrent Prolog is based on the notion of *read-only variables*. It in no way corresponds to the notion of *asking,* and a comparison will not be attempted.

AKL is closer in spirit to more recent languages, and in Sections 8.2.2, 8.2.3, and 8.2.4, AKL is compared to GHC, KL1, and PARLOG, respectively.

8.2.1 A Committed-Choice Computation Model

In this section, a computation model for a committed choice language closely resembling GHC is presented.

A program is a finite set of *guarded clauses.*

$$\begin{aligned}\langle\text{guarded clause}\rangle &::= \langle\text{head}\rangle \text{ :- } \langle\text{guard}\rangle \text{ ‘}|\text{’ } \langle\text{body}\rangle\\ \langle\text{head}\rangle &::= \langle\text{program atom}\rangle\\ \langle\text{guard}\rangle, \langle\text{body}\rangle &::= \langle\text{sequence of atoms}\rangle\end{aligned}$$

We keep the and-boxes of the CLP-model, which are augmented with sets of local variables, leave out the or-boxes, but instead introduce choice-boxes and guarded goals for guard computations.

$$\begin{aligned}\langle\text{goal}\rangle &::= \langle\text{local goal}\rangle \mid \langle\text{and-box}\rangle\\ \langle\text{and-box}\rangle &::= \textbf{and}(\langle\text{sequence of local goals}\rangle)^{\langle\text{constraint}\rangle}_{\langle\text{set of variables}\rangle}\\ \langle\text{local goal}\rangle &::= \langle\text{atom}\rangle \mid \langle\text{choice-box}\rangle\\ \langle\text{choice-box}\rangle &::= \textbf{choice}(\langle\text{sequence of guarded goals}\rangle)\\ \langle\text{guarded goal}\rangle &::= \langle\text{and-box}\rangle \text{ ‘}|\text{’ } \langle\text{statement}\rangle\end{aligned}$$

The model is a labelled transition system of the same type as that used in the AKL model. We use the constraint atom rule, the promotion rule, and the commit rule. Failure rules are not needed (but are harmless).

The single new rule is the *local forking* reduction rule

$$A \underset{\chi}{\overset{D}{\Rightarrow}} \textbf{choice}(\textbf{and}(G_1)^{\textbf{true}}_{V_1} \mid B_1, \ldots, \textbf{and}(G_n)^{\textbf{true}}_{V_n} \mid B_n)$$

which unfolds a program atom A with its definition

$$A \text{ :- } G_1 \mid B_1, \ldots, A \text{ :- } G_n \mid B_n$$

where the arguments of A are substituted for the formal parameters, and the local variables of the *i*th clause are replaced by the variables in the set V_i. The sets V_i are chosen to be disjoint from each other and from all other sets of local variables in the context χ.

8.2.2 GHC

GHC was proposed by Ueda as a rational reconstruction of PARLOG, Concurrent Prolog, and similar languages at the time [Ueda 1985].

A GHC program consists of *guarded clauses* of the following form.

h :- $g_1, \ldots, g_m$ | $b_1, \ldots, b_n$.

Each of the components (h, g_i, b_i) is an *atom*, of the form

$p(t_1, \ldots, t_k)$

where t_i are *terms*, expressions built from variables, numbers, constants, and constructors, and p is an alpha-numeric symbol, as in AKL atoms.

It should be apparent that GHC is a syntactic subset of AKL with rational tree constraints.

The comparison can be taken further in that we may describe GHC computations in terms of their difference from AKL computations. In fact, an AKL program written in the GHC subset will execute almost as prescribed by the GHC definition. The same results will be produced, but possibly (for contrived programs) at a higher or a lower cost, as discussed below.

The most obvious difference is that GHC avoids binding external variables. Roughly, a binding X=t is only made visible if X is a local variable. If X is external, the binding is said to *suspend*, waiting for X to become bound in an external environment. The advantage of this restriction is that no local binding environments have to be maintained, but there are also disadvantages as discussed below.

Local bindings will sometimes detect failure, for example in

p(X) :- X=1, X=2 | true.

In AKL an agent p(Z), with an unconstrained variable Z, would fail, whereas GHC would make it suspend. Worse is that GHC may randomly fail or suspend, depending on whether the guard in the following clause is executed from the left or from the right, respectively.

p(X) :- Y = 1, Y = 2, X = Y | true.

Although no sane programmer would write code like this, the situation could appear as the result of a deep guard execution, or as the result of a program transformation. This problem is almost immaterial in theoretical GHC, as the difference between suspension and failure plays no essential rôle for the semantics of a program. In KL1, where failure can be detected, the problem is more serious.

There are two alternative behaviours for GHC, which both seem to be in accordance with its definition: (1) If a binding operation is suspended, the whole guard is suspended, since the computation cannot be completed until a corresponding external variable has been produced. (2) Only the binding operation is suspended; the other agents in the guard may continue.

Consider a GHC program of the following form.

```
p(X, Y) :- X = a, q(Y, X, Z) | r(Z).
p(X, Y) :- … .
```

The first guard not only awaits a binding, but also performs a computation. Assume that the execution of q is time-consuming and that the binding X = a is not yet available when p is first entered.

In some implementations of behaviour (1) the binding will be tried before, or very early in, the execution of q. Then most of q will suspend and will only be available for execution after the binding has been produced from elsewhere. This can lead to fewer computation steps than in AKL, where it is possible that steps are spent on q before trying other clauses, which may very well be applicable immediately. However, the program can execute slower in a parallel implementation of this kind than in a parallel implementation of AKL if processors are in good supply. In AKL, the producer of the binding X = a and q may execute in parallel. Behaviour (1) would make them execute in sequence, which could take longer time.

Behaviour (2) is closer to that of AKL, but it can be more wasteful, as follows. Again consider the above example. If, while q is being executed and no other clause is applicable, a conflicting binding X = b is suddenly produced by q, then the AKL guard would fail. GHC (2) would not detect the conflict, and work would be wasted on q.

If nothing else, this discussion shows that the differences between AKL and GHC are of the same calibre as the differences between alternative interpretations of GHC itself. It has even been claimed that the behaviour of the corresponding subset of AKL is indeed a viable interpretation of the GHC definition.

8.2.3 KL1

KL1 is based on Flat GHC (FGHC), a theoretically cleaner language, which, in its turn, is a special case of GHC [Ueda and Chikayama 1990; Chikayama 1992]. The “flatness” refers to a restriction on the guard part of a clause, which may not contain program atoms defined by the user program.

Thus, in KL1, in a clause of the form

h :- $g_1, …, g_m$ | $b_1, …, b_n$.

the atoms g_i in the guard are restricted to (tree equality) constraints and certain built-in operations. This restriction will make the computation state flat. There will be no nested “choice-statements” as when the guard may contain arbitrary statements, and this simplifies the implementation considerably.

KL1 allows “pragma” of the form

% otherwise

and

% alternatively

which may be placed between the clauses in a definition.

The first, *otherwise*, means that the clauses following it should only be tried if the guards in all preceding clauses fail. The effect is similar to that of conditional choice in AKL, and even more similar to the sequential clause operator of PARLOG, for which a mapping into AKL is shown in the next section. When primitives that detect failure are introduced, such as otherwise, the order-dependence of the GHC suspension rule has to be taken into account. For KL1 the solution is to impose a sequential left-to-right execution order in guards, which gives predictable, if not ideal, behaviour.

The second, *alternatively*, means that clauses following it should only be tried if the guards in all preceding clauses have failed or are *currently* suspended at the time of attempting reduction. There is no similar construct in AKL.

KL1 also provides new high-level constructs, such as the *sho-en*, which is similar to the *engine* concept proposed for AKL.

A low-level facility of KL1 is the MRB optimisation, which allows data structures such as arrays to be updated in place in constant time if they are single-referenced. As only one reference may be held at any given instant, access is serialised. By splitting an array into two disjoint parts, the two may be updated in parallel, to be joined at a later time. AKL provides more flexible models of mutable data based on ports (Chapter 7).

8.2.4 PARLOG

PARLOG provides a number of constructs of potential interest [Clark and Gregory 1986; Gregory 1987]. Here we discuss Kernel PARLOG, the standard form without mode declarations and with explicit head unification. We will not discuss the don't know extension, the simple functionality of which is entirely subsumed by AKL. Familiarity with PARLOG is assumed.

A PARLOG definition is a sequence of clauses, grouped by *parallel* (“.”) and *sequential* (“;”) *clause composition* operators.

$$\begin{aligned}\langle\text{def}\rangle &::= \langle\text{clause}\rangle \mid (\langle\text{def}\rangle \,.\, \langle\text{def}\rangle) \mid (\langle\text{def}\rangle \,;\, \langle\text{def}\rangle)\\ \langle\text{clause}\rangle &::= \langle\text{atom}\rangle \mathrel{<\!-} \langle\text{goal}\rangle : \langle\text{goal}\rangle\\ \langle\text{goal}\rangle &::= \langle\text{atom}\rangle \mid (\langle\text{goal}\rangle, \langle\text{goal}\rangle) \mid (\langle\text{goal}\rangle \,\&\, \langle\text{goal}\rangle)\end{aligned}$$

The symbols “<-” and “:” correspond to “:-” and “ | ” in AKL. The operators “,” and “&” are *parallel* and *sequential conjunction*, respectively. The former corresponds to the composition statement of AKL; the latter is discussed below.

PARLOG in AKL

The difference between parallel and sequential clause composition is similar to that between committed and conditional choice in AKL. Thus, a definition

H <- G_1 : B_1.
…
H <- G_n : B_n.

which uses only “.”, can be translated to

H^* :- G_1^* | B_1^*.
…
H^* :- G_n^* | B_n^*.

where * denotes a mapping from PARLOG goals to AKL statements, and a corresponding definition

H <- G_1 : B_1;
…
H <- G_n : B_n.

which uses only “;”, can be translated to

H^* :- $G_1^* \rightarrow B_1^*$.
…
H^* :- $G_n^* \rightarrow B_n^*$.

However, when parallel and sequential clause composition operators are arbitrarily nested, the translation becomes less straightforward. The following is a mapping from definitions with arbitrary nesting of the clause composition operators to corresponding committed choice and conditional choice statements. It is awkward, and never needed for “real” programs, but included for the sake of completeness.

Number the guards and bodies of the clauses c_1 to c_n in the (Kernel) PARLOG definition g_1 to g_n and b_1 to b_n, correspondingly. The clause c_i has local variables $X_{i,1}$ to X_{i,k_i}. We assume that all heads are equal (with different variables as arguments); let H represent these.

A PARLOG definition is mapped to an AKL definition with a single commit clause. The guard of this clause computes a continuation which identifies chosen body, and which contains local variables shared between guard and body.

$$\begin{aligned} D^* \Rightarrow H^* :- D^{\#(1)} \mid \quad & (X_{1,1}, \ldots, X_{1,k_1} : B_0 = \mathrm{cont}(1, X_{1,1}, \ldots, X_{1,k_1}) \rightarrow b_1^* \\ & ; \ldots \\ & ; X_{n,1}, \ldots, X_{n,k_n} : B_0 = \mathrm{cont}(n, X_{n,1}, \ldots, X_{n,k_n}) \rightarrow b_n^*). \end{aligned}$$

The following rules map sequential and parallel clause composition to conditional and committed choice, respectively.

$$\begin{aligned} & (D_1 ; \ldots ; D_m)^{\#(n)} \Rightarrow \\ & \qquad (B_n : D_1^{\#(n+1)} \rightarrow B_{n-1} = B_n \\ & \qquad ; \ldots \\ & \qquad ; B_n : D_m^{\#(n+1)} \rightarrow B_{n-1} = B_n) \end{aligned}$$

$$
\begin{aligned}
&(D_1 \,.\, \ldots \,.\, D_m)^{\#(n)} \Rightarrow \\
&\qquad (\, B_n : D_1{}^{\#(n+1)} \mid B_{n\text{-}1} = B_n \\
&\qquad ;\, \ldots \\
&\qquad ;\, B_n : D_m{}^{\#(n+1)} \mid B_{n\text{-}1} = B_n \,)
\end{aligned}
$$

If a guard succeeds, it returns a continuation for its corresponding body.

$$C_i{}^{\#(n)} \Rightarrow (\, X_{i,1}, \ldots, X_{i,k_i} : g_i{}^{*} \mid B_{n\text{-}1} = \mathrm{cont}(i, X_{i,1}, \ldots, X_{i,k_i})\,)$$

A complete mapping of sequential conjunction requires distributed termination techniques, for example *short-circuiting* (cf., [Shapiro 1986a]). This is demonstrated by the interpreter in the next section. Certain restricted cases of sequencing G_1 & G_2 can be expressed as (G_1 ? G_2) using nondeterminate choice. The difference is that the guard G cannot communicate with other agents until it has been completely evaluated.

PARLOG Interpreter in AKL

The above style of translation of sequential and parallel composition can also be expressed in terms of an interpreter for (a subset of) Kernel PARLOG in AKL. The sequential and parallel clause composition operations are represented by the constructors seq(C,D) and par(C,D), respectively. Here is also included a mapping of sequential conjunction in terms of short-circuiting. Observe the need for a special treatment of primitives.

```
parlog(true, C0, C) :-
    → C0 = C.
parlog((P,Q), C0, C) :-
    → parlog(P, C0, C1),
      parlog(Q, C1, C).
parlog((P&Q), C0, C) :-
    → parlog(P, D0, D),
      ( D0 = D | parlog(Q, C0, C) ).
parlog(H, C0, C) :-
      is_primitive(H)
    → primitive(H, C0, C).
parlog(H, C0, C) :-
      definition(H, D)
    → try(D, H, B),
      parlog(B, C0, C).

try(seq(C,D), H, B) :-
    → ( B0 : try(C, H, B0) → B = B0
      ; B0 : try(D, H, B0) → B = B0 ).
try(par(C,D), H, B) :-
    → ( B0 : try(C, H, B0) | B = B0
      ; B0 : try(D, H, B0) | B = B0 ).
```

```
try(clause(C), H, B) :-
        instance(C, (H <- G : B0)),
        parlog(G, _, _)
    → B = B0.
```

An example encoding of a definition (concat) follows.

```
definition(concat(X,Y,Z), Def) :-
    → Def = par(concat_1(X,Y,Z), concat_2(X,Y,Z)).

instance(concat_1(X,Y,Z), Clause) :-
    → Clause = (concat(X,Y,Z) <- X = [] : Z = Y).
instance(concat_2(X,Y,Z), Clause) :-
    → Clause = (concat(X,Y,Z) <- X = [E | X1] : Z = [E | Z1], concat(X1,Y,Z1)).

is_primitive((X = Y)) :- → true.

primitive((X = Y), C0, C) :-
    → X = Y, C = C0.
```

In the case of (equality) constraints, sequencing has no meaning in AKL.

Discussion

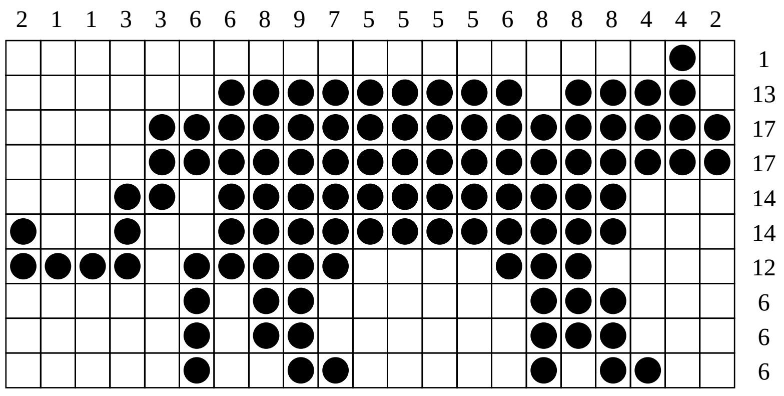
One might ask if it is necessary or even desirable to have general sequencing as a primitive in this kind of language. One common use is to sequence side effect operations, which have been added to the language in place of properly integrated support for interoperability. Other uses are to improve efficiency and resource allocation by explicit tampering with the execution order.

If a producer and a consumer can execute sequentially, they can be sequenced with a sequencing operator. Efficiency can be gained; there is no context switching between processes. If the consumer cannot take any action until its corresponding producer has terminated, efficiency is always gained. Efficiency can also be lost; nothing will be consumed while the producer is working, and the working set will grow, decreasing locality and increasing the time spent on garbage collections. Of course, potential parallelism is also lost.

AKL does not provide sequencing, since it does not add to the expressiveness of the language and there is good hope that sufficient efficiency can be achieved without such interventions by the programmer, adjusting this, adjusting that.

For example, unless the language requires fairness, there is no need for unnecessary context switching between processes. On the other hand, the language implementation is free to, for example, switch to the consumer when memory consumption of the producer exceeds a certain threshold.

Finally, we can note that a commercially available implementation of what is essentially Flat PARLOG is the language Strand [Foster and Taylor 1989, 1990]. A further development of Strand is PCN [Foster and Tuecke 1991].



Upper-left to lower-right diagonals 0, 0, 0, 0, 0, 1, 2, 5, 6, 4, 4, 6, 7, 7, 8, 8, 5, 6, 6, 6, 6, 5, 5, 3, 3, 1, 1, 1, 0, 0

Upper-right to lower-left diagonals 0, 1, 2, 3, 3, 3, 2, 4, 5, 5, 7, 9, 9, 6, 6, 5, 4, 5, 5, 4, 5, 4, 2, 2, 2, 2, 1, 0, 0, 0

Figure 8.3. A sample scanner grid

8.3 AKL VS. CONSTRAINT LOGIC PROGRAMMING

In this section, it is demonstrated by examples how AKL goes beyond basic constraint logic programming (CLP).

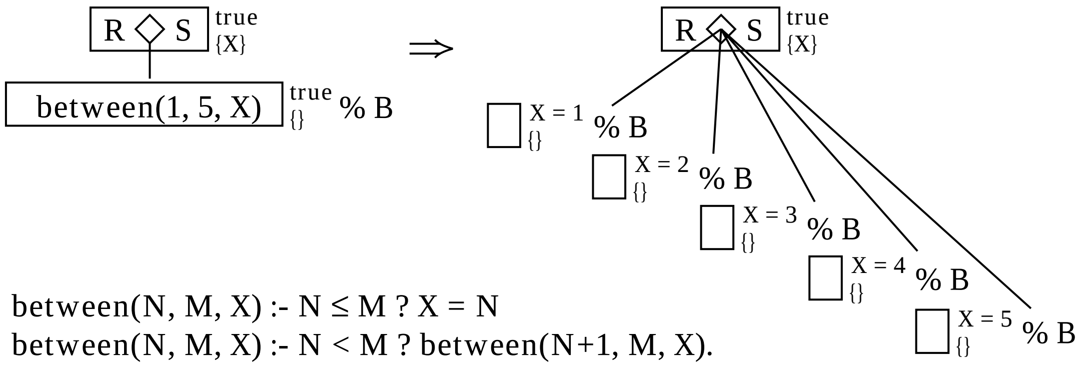
When constraint programming in AKL, the underlying constraint system serves as the basic terminology, which may stand for a lesser or a greater portion of the solution depending on its strength. But AKL may also be used to program constraint solving. For example, finite domain constraint solving can be expressed in AKL with the constraint system of trees (implied by work by Bahgat and Gregory [1989] and Gregory and Yang [1992]). Whereas the logical content of the corresponding constraints could also be expressed in, e.g., a constraint logic programming (CLP) language, the necessary constraint propagation would not be achieved.

In Section 3.5, a solution of the N-Queens problem illustrated programmable constraint satisfaction techniques. It employed a combination of the short-circuit and the mutual exclusion variable techniques to express the *exactly-one* constraint, and the propagation effect achieved corresponds to that of *forward checking* in CLP [Van Hentenryck 1989].

To show techniques applicable also in many other situations, a new example is used, called the Scanner[1]. The problem is to reveal the contents of a grid. Each square may be filled or non-filled. The input is the number of filled squares along each row, column, and diagonal. In general, there may be no, one, or several solutions for a given input. Figure 8.3 shows a sample input and solution.

[1] The Scanner was implemented in AKL by Johan Montelius. It was originally described by A.K. Dewdney in Scientific American (September, 1990, p. 124).

Analogously to how the exactly-one and at-most-one constraints were used for the N-Queens problem, the Scanner problem can be reduced to an *exactly-N* constraint, which is used to express all constraints.



```
between(N, M, X) :- N ≤ M ? X = N
between(N, M, X) :- N < M ? between(N+1, M, X).
```

Figure 8.4. Local enumeration of possible values

The constraint exactly_n(N, K, L) is read “N out of a total K elements in the list L are 1; the rest are 0”. The propagation effect desired is that whenever N elements are known to be 1, the rest are known to be 0, and whenever K-N are known to be 0, the rest are known to be 1. In the grid in Figure 8.3, we would first know that there are no filled squares in column 1; we would then know that the rest of the squares in the top-left to bottom-right diagonal were filled; we would then know that the rest of the squares in the right-most and bottom-most rows were blank; and so on.

This constraint cannot be expressed in the same simple way as the exactly-one constraint. Two techniques will be shown, that of *local execution* and the very general *monitor-controller* technique. But first we specify the relation in almost CLP style and discuss its properties.

```
exactly_n(0, 0, []).
exactly_n(N, K, [1|L]) :-
        N > 0
    ?   exactly_n(N-1, K-1, L).
exactly_n(N, K, [0|L]) :-
        K > N
    ?   exactly_n(N, K-1, L).
```

Logically, it should be clear that the above program expresses the exactly-N constraint. It does not, however, perform propagation as desired. As written, an exactly_n call suspends if the first element of the list is not given. The values of other elements cannot be taken into account, and the two desired forms of propagations are not performed.

Instead of suspending, we could of course consider generating all possible lists. *Local execution* means enumerating alternatives for a variable (or a group of variables) locally in the guard of a nondeterminate choice statement (Figure 8.4). Propagation will be automatic by the pruning of failed alternatives. When the choice becomes determinate, that solution will be promoted, and propagation will occur.

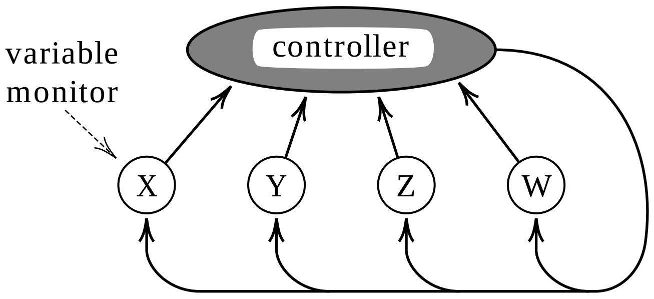


Figure 8.5. Monitors and controller

The exactly-N constraint can be expressed as follows using the local execution technique.

```
exactly_n(N, K, L) :-
        exactly_n_aux(N, K, L)
    ?   true.

exactly_n_aux(0, 0, L) :-
    ?   L = [].
exactly_n_aux(N, K, L0) :-
        N > 0
    ?   L0 = [1 | L],
        exactly_n_aux(N-1, K-1, L).
exactly_n_aux(N, K, L0) :-
        K > N
    ?   L0 = [0 | L],
        exactly_n_aux(N, K-1, L).
```

Again, it should be clear that the above program expresses the exactly-N constraint, the only differences being the placement of the goal in the guard and the moving of constraints on the list to the body. It does perform propagation as desired, but at a great cost. If nothing is known about the list, $\binom{N}{K}$ alternative lists are created. Thus, in this case, local execution should typically not be used (but see Section 8.3.2 for a case where it can).

Monitor-controller means having a monitor process for each variable, which reports back to a controller when certain constraints are told on its variable (Figure 8.5). The controller is able to propagate information back to the monitors, which are usually addressed collectively by broadcasting.

The exactly-N constraint can be expressed as follows using the monitor-controller technique.

```
exactly_n(N, K, L) :=
        open_port(P, S),
        spawn_monitors(L, P, B),
        controller(S, N, K, B).
```