- *side effects* such as input and output operations on files and operations on the internal database
- *metalogical* operations such as var/1

These extensions have no interpretations in terms of SLD-resolution.

We will discuss how Prolog programs with cut and side effects can be executed in AKL. Metalogical operations are not provided for; why will be discussed. A familiarity with Prolog will be assumed (e.g., as can be acquired from [Clocksin and Mellish 1987], [Sterling and Shapiro 1986], or [O'Keefe 1990]).

### *8.1.4 Cut*

The *cut* pruning operation in Prolog is similar to conditional choice in AKL. For example, a Prolog definition

```
p(a, Y) :- !, q(Y).
p(X, Y) :- r(Y).
```

may be translated to AKL as

```
p(a, Y) :- → q(Y).
p(X, Y) :- → r(Y).
```

if in each of its uses, the first argument is known to be equal to ‘a’ or known to be different from ‘a’. In this case, cut may be performed without having added constraints to the arguments of p/2 calls. This is called *quiet pruning*. If, on the other hand, the first argument is unknown, the Prolog cut operation would still take effect, even though it is not known that the first argument is equal to ‘a’. This is called *noisy pruning*. The corresponding AKL program would suspend in this case. For the purpose of translation into AKL, it is desirable to transform, if possible, programs using noisy pruning to programs using only quiet pruning. For example, if the above Prolog program is always used with a variable as the first argument, it can be safely transformed into

```
p(X, Y) :- !, X = a, q(Y).
p(X, Y) :- r(Y).
```

where the use of cut is quiet. If there are occurrences of p/2 atoms in the program that are always used in one way, and other occurrences that are always used in the other, then two different definitions can be used to make both cases quiet. The full functionality of noisy cut pruning is only required if the same occurrence is used in both ways. (An experimental extension of AKL with a (*noisy*) *cut choice* statement, with the ability to perform noisy pruning, is described in Section 4.7.2.)

It is possible to make translation from Prolog into AKL completely automatic along the lines outlined in this section using data-flow analysis by abstract interpretation [Bueno and Hermenegildo 1992]. The noisy cut extension is then used when no other translation is possible. The use of noisy cut also involves using means to sequentialise the program to avoid back-propagation of values that violates Prolog semantics. Bueno and Hermenegildo also observed that the performance of Prolog programs was improved by transforming from noisy to quiet cut.

Although similar to conditional choice, cut in Prolog may be used quite arbitrarily, e.g., as in

```
p :- q, !, r, !, s.
p :- t.
p :- u, !.
```

To correspond more closely to AKL definitions, each clause should contain exactly one cut or no clause should contain cut. We first observe that the repeated use of cut in the first clause can be factored out, as in

p :- q, !, $p_1$.

with an auxiliary definition

$p_1$ :- r, !, s.

We then factor out a cut-free definition containing the second clause, and a third cut-definition with the third clause. The resulting program is

p :- q, !, $p_1$.
p :- !, $p_2$.

$p_1$ :- r, !, s.

$p_2$ :- t.
$p_2$ :- $p_3$.

$p_3$ :- u, !.

which, if the uses of cut are quiet, corresponds to the AKL definition

p :- q $\rightarrow$ $p_1$.
p :- $\rightarrow$ $p_2$.

$p_1$ :- r $\rightarrow$ s.

$p_2$ :- ? t.
$p_2$ :- ? $p_3$.

$p_3$ :- u $\rightarrow$ true.

Such awkward translations are rarely necessary in practice. Our experience of translating Prolog programs is that elegant translations are usually available in specific cases. Typically, Prolog definitions correspond more or less directly to a conditional choice or to a nondeterminate choice. It should be noted that there is no fully general translation of cut inside a disjunction. However, it is the opinion of the author that this abominable construct should be avoided anyway.

Unfortunately, the quietness restriction also makes some pragmatically justifiable programming tricks impossible in AKL that are possible in Prolog. These tricks depend on the sequential flow of control, and the resulting particular instantiation patterns of the arguments of a cut-procedure in specific execution states. There are usually work-arounds that do not involve noisy pruning. However, in some cases, the translation is quite non-trivial, and cannot be readily automated.

A typical case is presented together with suggestions for alternative solutions in AKL. The principles underlying these alternative solutions can be adapted to other similar cases. The following lookup/3 definition in Prolog relies on noisy cut to add a key-value pair automatically at the end of a partially instantiated association list if a pair with a matching key is not found.

```
lookup(K, V, [K=V1 | R]) :- !, V = V1.
lookup(K, V, [_ | R]) :- lookup(K, V, R).
```

Clearly, the corresponding AKL definition would only be able to find existing occurrences of the key. There are several work-arounds for this problem. It is for example possible to manage a complete list of key-value pairs, as in the following AKL program.

```
lookup(K, V, L, NL) :-
        lookup_aux(K, V1, L)
    →   V = V1,
        NL = L.
lookup(K, V, D, ND) :-
    →   ND = [K=V | D].

lookup_aux(K, V, [K=V1 | D]) :-
    →   V = V1.
lookup_aux(K, V, [X | D]) :-
    →   lookup_aux(K, V, D).
```

Note that the AKL definition corresponding to the above Prolog program is used as an auxiliary definition. The cost of searching and adding new elements remains the same.

However, when using this program, it is necessary to pass the list around more explicitly than in the Prolog solution. Note also that access is serialised. A list cannot be updated concurrently. The following solution allows concurrent lookups. It uses an incomplete list as in the Prolog solution, but access to the dictionary is managed by a dictionary server which synchronises additions to the tail of the list. The server is accessed through a port.

```
alist(P) :-
        open_port(P, S),
        alist_server(S, D, D).

alist_server([lookup(K, V) | S], D, T) :-
    →   lookup(K, V, D, T, NT),
        alist_server(S, D, NT).
alist_server([], D, T) :-
    →   true.
```

```
lookup(K, V, [K1=V1 | R], T, NT) :-
        K = K1
    |   V = V1,
        T = NT.
lookup(K, V, [K1=V1 | R], T, NT) :-
        K ≠ K1
    |   lookup(K, V, R, T, NT).
lookup(K, V, T, T, NT) :-
    |   T = [K=V | NT].
```

The lookup definition is like the previous ones but for the last two arguments and the third clause. The extra arguments hold the tail of the list and the new tail of the list. If nothing is inserted, the old tail is returned. The third clause detects that the list is equal to its tail, inserts a new pair, and returns the new tail of the list. The definition has to be don't care nondeterministic, since the guards of the preceding clauses have not been refuted. Each lookup request is given the new tail of the preceding one, thus serialising updates, but permitting concurrent lookups.

We can note that this could also have been an FGHC program, but for the fact that FGHC implementations usually do not consider variable identity as a quiet case, and therefore the third lookup-clause will not recognise the uninstantiated tail as intended.

The above program can be extended to perform a checking lookup which does not add the new element, even though the list ends with a variable. In Prolog a metalogical primitive would be needed to achieve the same effect.

The incomplete structure technique is more useful when the lookup-structure is organised as a tree. Unless sophisticated memory management techniques are used (such as reference counting, producer-consumer language restrictions, or compile-time analysis), alternative solutions with complete structures will require O(log(N)) allocated memory for each new addition to the tree, but they are also less satisfactory because they require that access is serialised.

The following Prolog program will only allocate the new node.

```
treelookup(K, V, t(K, V1, L, R)) :- !, V = V1.
treelookup(K, V, t(K1, _, L, R)) :- K < K1, !, treelookup(K, V, L).
treelookup(K, V, t(_, _, L, R)) :- treelookup(K, V, R).
```

In AKL, the tree can be represented by a tree of processes, e.g., as shown in Section 3.2.3. It can be argued that this is less efficient than the Prolog solution, but a compilation technique for FGHC programs that optimises such programs, *message-oriented scheduling*, suggests that this inefficiency is not an inherent problem [Ueda and Morita 1992].

Finally, we observe that the troublesome *negation as failure,* expressed as

```
not(P) :- call(P), !, fail.
not(_).
```

in Prolog, is always *sound* in AKL when expressed as

```
not(P) :- call(P) → fail.
not(_) :- → true.
```

(see Section 4.8.2). A more complete, and still sound, definition of negation is achieved by wrapping the called goal in a committed choice, as in

```
not(P) :- (call(P) | true) → fail.
not(_) :- → true.
```

It is now possible to call

```
        not(member(X, [a, X, b]))
```

using member defined as in Section 2.5, and get the desired failure.

### *8.1.5 Side Effects*

It is possible to model definite clauses with side effects in AKL, while still interpreting programs in a true “metainterpreter” style, mapping object-level nondeterminism to metalevel nondeterminism. The program as such is probably not very useful, but it illustrates the versatility of AKL, and also introduces programming techniques with wider applicability.

The program makes use of a technique related to that used for the constant delay multiway merger defined in Section 7.3.2.

An ordered bagof agent encapsulates the or-tree formed by the interpreter. Solutions are communicated to a server process, which is also in charge of side effects. In the example interpreter, the only possible side effects are reading and writing a value held in the server, but this can be extended, e.g., to a scheme reminiscent of assert/retract or to include I/O.

When the interpreter wishes to perform a side effect, it communicates a special “solution” to bagof which is then sent to the side effect server. This solution is either a read or a write message. In a write, the value to be written is simply given as its argument.

Read is more tricky. Since the bagof collection operation renames local variables, a read result cannot easily be returned using local variables. The read operation must have access to a *unique* global variable that is sent to the server and on which the result can be returned. This is achieved as follows. The interpreter is given an extra external variable as an argument. Upon every branching performed, a redundant “solution” is returned, which communicates to the server that a split has been performed, sending its external variable as an argument. The server then binds this variable to a structure split(_,_). The two new branches select one new external variable each, and continue. The read operation sends the current external variable. The result is returned by binding the external variable to a structure read(V,_), where the first argument is the value read, and the second argument is a new external variable for future communications.

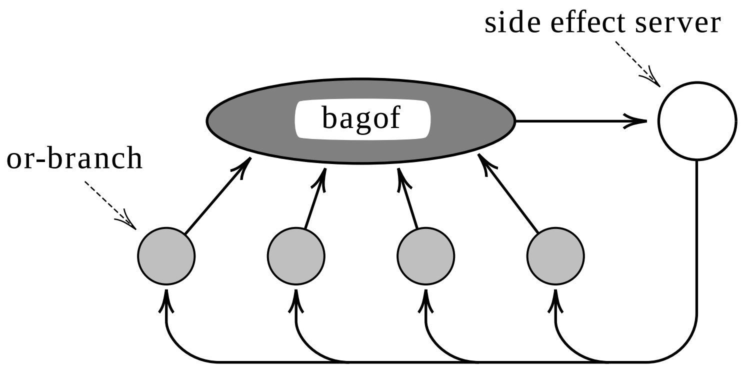

Figure 8.1. Definite clause interpreter with side effects

The complete interpreter follows. As explained above, the interpreter consists of a bagof and a server process.

```
solve :-
        bagof(X, solve(X, _), Y),
        server(Y, []).

solve(X, E) :-
        G = ⟨goal to be called⟩,
        solve([G], X, G, E).
```

The server deals with the different forms of “solutions” that may be produced: write, read, split, and a real solution.

```
server([], _) :-
    → true.
server([write(U) | Xs], V) :-
    → server(Xs, U).
server([read(U) | Xs], V) :-
    → U = read(V,_), server(Xs, V).
server([split(E) | Xs], V) :-
    → E = split(_, _), server(Xs, V).
server([solution(_) | Xs], V) :-
    → server(Xs, V).
```

The interpreter catches the special cases, dispatching to special procedures for read and write, and to clause trying for other goals.

```
solve([], X, G, E):-
    → X = solution(G).
solve([write(U) | As], X, G, E) :-
    → solve_write(U, As, X, G, E).
solve([read(U) | As], X, G, E) :-
    → solve_read(U, As, X, G, E).
solve([A | As], X, G, E) :-
        clauses(A, Cs)
    → try(A, Cs, As, X, G, E).
```

Write and read send messages to the server in special “solutions”. The read message, with the value and a new variable, is received in the second branch.

```
solve_write(U, As, X, G, E) :-
    ?  X = write(U).
solve_write(U, As, X, G, E) :-
    ?  solve(As, X, G, E).

solve_read(U, As, X, G, E) :-
    ?  X = read(E).
solve_read(U, As, X, G, E) :-
    ?  receive_read(E, U, As, X, G).

receive_read(read(V,E), U, As, X, G) :-
    →  U = V,
       solve(As, X, G, E).
```

A split is performed if there are two or more clauses to try. When splitting, a message is sent to the server, to allow it to generate new external variables that will correspond to the E's in the second and third clauses.

```
try(A, [_,_|_], As, X, G, E) :-
    ?  X = split(E).
try(A, [(A:-Bs)|_], As, X, G, split(E,_)) :-
    ?  append(Bs, As, ABs),
       solve(ABs, X, G, E).
try(A, [_,C|Cs], As, X, G, split(_,E)) :-
    ?  try(A, [C|Cs], As, X, G, E).
```

A limitation of the scheme presented here compared to Prolog is that read terms containing variables may not be manipulated freely. Upon writing, a term is promoted to the external environment, making any variable external. Such variables may not be bound in the interpreter. Effectively, this means that the read and write operations are restricted to ground terms.

### *8.1.6 Metalogical Operations*

Prolog supports a number of so called *metalogical* operations that cannot be explained in terms of the basic computation model. All have in common that they regard unbound variables as objects. They can establish that a variable is unbound, test variables for equality, and compare them using the standard term ordering. One such notorious operation is var/1, which tests whether a given variable is unbound.

Such an operation does not rhyme well with concurrency. It destroys the property that all conditions are monotone. Once a guard is quiet or incompatible with its environment, this will continue to hold. The behaviour of var is anti-monotone, changing from success to failure if its argument is bound. Although pragmatically justifiable uses can be found, the very availability of such an operation encourages a poor programming style, and it is quite possible to do without it. For this reason, all operations of this kind have been omitted from AKL.

### *8.1.7 Prolog and Parallelism*

Considerable attention has been given to the topic of making Prolog programs run in parallel. Roughly, there are three basic approaches: *or-parallelism*, where the different branches in the search tree are explored in parallel [Lusk et al. 1988; Carlsson 1990; Karlsson 1992], *independent and-parallelism*, where goals may run in parallel if they do not disagree on shared variables [DeGroot 1984; Hermenegildo and Greene 1990], and *dependent and-parallelism*, where goals may be reduced in parallel, although they may potentially bind shared variables, as in the committed choice languages [Naish 1988; Santos Costa, Warren, and Yang 1991a; 1991b]. There are also various combinations of the above that await evaluation [Gupta et al. 1991; Gupta and Hermenegildo 1991; 1992].

The concurrency of AKL provides a potential for parallel execution. In the following sections, it is briefly discussed how concurrency (a potential for parallel execution) corresponding to the above three forms of parallelism may be identified in AKL programs.

When a computation is split into alternative computations by performing a nondeterminate choice, this forms a tree in a manner quite analogous to the search tree formed by Prolog execution.

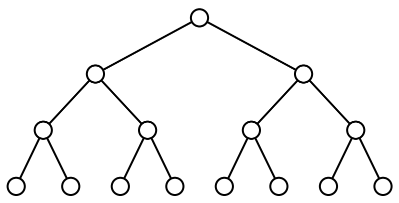
Figure 8.2. An or-tree

The different branches are quite independent, and computation steps may be performed concurrently. Clearly, this provides AKL with a potential for *or-parallel execution*.

Execution mainly consists of unfolding goals, some pruning of alternatives in choices, followed by a choice of the single remaining alternative. The only interaction between such goals is through shared variables. This corresponds to committed choice execution and can be exploited as *dependent and-parallelism* in the usual manner.

Consider the well-known quicksort program.

```
qsort([], R, R).
qsort([X | L], R0, R) :-
        partition(L, X, L1, L2),
        qsort(L1, R0, [X | R1]),
        qsort(L2, R1, R).
```

partition([], C, [], []).
partition([X | L], C, [X | $L_1$], $L_2$) :-
        X < C
    ?   partition(L, C, $L_1$, $L_2$).
partition([X | L], C, $L_1$, [X | $L_2$]) :-
        X >= C
    ?   partition(L, C, $L_1$, $L_2$).

Execution of a goal qsort([2,3,1], L, []) will be completely determinate, and the AKL is able to extract concurrency as follows. The goals have some arguments suppressed for the sake of brevity, and the goals that are determinate are underlined. Determinate goals are reduced in one step.

$$\begin{array}{lllllll}
\underline{\mathrm{qs}([2,3,1])} & & & & & & \\
\underline{\mathrm{p}([3,1])}, & & \mathrm{qs}(L_1), & & \mathrm{qs}(L_2) & & \\
\underline{\mathrm{p}([1])}, & & \mathrm{qs}(L_1), & & \underline{\mathrm{qs}([3 \mid L_3])} & & \\
\underline{\mathrm{p}([])}, & \underline{\mathrm{qs}([1 \mid L_4])}, & & & \underline{\mathrm{p}([])}, & \mathrm{qs}(L_5), & \mathrm{qs}(L_6) \\
 & \underline{\mathrm{p}([])}, & \mathrm{qs}(L_7), & \mathrm{qs}(L_8), & & \underline{\mathrm{qs}([])}, & \underline{\mathrm{qs}([])} \\
 & & \underline{\mathrm{qs}([])}, & \underline{\mathrm{qs}([])} & & & \\
\square & & & & & &
\end{array}$$

Quite a lot of potential parallelism is extracted.

Various forms of execution corresponding to the *independent and-parallelism* exploited for Prolog are conceivable. To emphasise the similarity with Prolog, it is shown how pure definition clauses can be translated in such a way that a potential for independent and-parallelism appears.

Assume that in the clause

p(X) :- q(X, Y), r(X, Z), s(Y, Z).

the goals q and r are found to be independent. In the context of restricted and-parallelism, this means that the program that uses p calls it with an argument X such that neither of the goals q or r will instantiate X further. The following translation into AKL enables independent parallel execution of q and r as soon as X is sufficiently instantiated by its producers.

p(X) :- true ? $q_1$(X, Y), $r_1$(X, Z), s(Y, Z).

$q_1$(X, Y) :- q(X, $Y_1$) ? Y = $Y_1$.

$r_1$(X, Z) :- r(X, $Z_1$) ? Z = $Z_1$.

By putting the goals in guards and extracting the output argument, unless the goals attempt to restrict X, all computation steps are always admissible, including choice splitting.

This style of translation can make use of the tools developed for restricted and-parallelism, such as compile-time analysis of independence, making it completely automatic [Bueno and Hermenegildo 1992].

### *8.1.8 The Basic Andorra Model*

The *Basic Andorra Model* (BAM) is a control strategy for the definite clause computation model with the appealing property of giving priority to deterministic work over nondeterministic work [Haridi and Brand 1988]. The BAM was first proposed by D. H. D. Warren at a GigaLIPS meeting in 1987, and was also discovered independently by Smolka [1993], who dubbed it *residuation* and described it for general constraints.

The BAM divides a computation within and-boxes into *determinate* and *nondeterminate phases*. First, all program atoms with at most one candidate (*determinate* atoms) are reduced during the determinate phase. Then, when no determinate atom is left, an atom is chosen for which all candidates are tried; this is called the nondeterminate phase. The computation then proceeds with a determinate phase on each or-branch.

The BAM has a number of interesting properties.

Firstly, all reductions in the deterministic phase may be executed in parallel, thereby extracting implicit dependent and-parallelism from pure definite clause programs, as in NUA-Prolog [Palmer and Naish 1991]. Andorra-I provides both dependent and- and or-parallelism on the Sequent Symmetry [Santos Costa, Warren, and Yang 1991a; 1991b].

Secondly, the notion of determinacy is a form of synchronisation. While data is being produced during the determinate phase, consumers of this data are unable to run ahead (since this would make them nondeterminate). This makes it possible to program with a notion of concurrent processes.

Thirdly, executing the determinate goals first reduces the search space, and thus, in general, the execution time. Goals can fail early, and the constraints produced by a reduction can reduce the number of alternatives for other goals. This is very relevant for the coding of constraint satisfaction problems [Bahgat and Gregory 1989; Haridi 1990; Yang 1989; Gregory and Yang 1992].

In AKL, the principle underlying the BAM has been generalised to a language with deep guards, and is embodied in the stability condition. However, the BAM itself is available as a special case. Below is shown a simple translation from program clauses into AKL. The translated programs will behave exactly as dictated by the BAM.

By putting the constraints in the guard of a wait-clause, local execution in the guard will establish whether a clause is a candidate. Other rules have priority over choice splitting, and therefore execution in AKL will conform to the BAM.

A program clause

```
A :- C, B.
```

where C is a constraint, is translated into a corresponding AKL clause as

```
A :- C ? B.
```

according to the above suggested scheme.