

An Overview of the Andorra Kernel Language*

Torkel Franzén Seif Haridi Sverker Janson

Swedish Institute of Computer Science[†]

March 20, 1992

1 Introduction

The Andorra Kernel Language (AKL) is a general combination of search-oriented nondeterministic languages, such as Prolog, and the process-oriented committed-choice languages, such as GHC [10, 8, 9]. For an introduction to the language from this perspective, see [7]. For an extensive formal treatment, see [4].

Although the Prolog programming paradigm is provided, AKL does not provide the exact operational semantics of the Prolog cut, nor does it provide Prolog style side-effects. Automatic translations of large subsets of Prolog into AKL are possible by using dataflow analysis [2]. On the other hand, GHC is more or less a sublanguage of AKL. This is not surprising, since both GHC and AKL are concurrent logic programming languages, designed to have a large potential for parallel execution.

AKL provides new control principles for search programs, some of which are inherited from the Andorra model [1, 12], and some of which are entirely new [7]. For example, it is possible to use don't know nondeterminism within concurrent reactive processes. This provides opportunities for distributed problem solving. It is also possible to structure search programs beyond plain left-to-right depth-first search.

In the following sections, we will give examples which illustrate capabilities of AKL beyond those of Prolog. We then describe the language and its computation model quite formally to be able to discuss a couple of technical points, and to be able to state soundness and completeness results.

2 Two Simple Examples

We will show two simple AKL examples in this section. The first example illustrates constraint propagation techniques inherited from the Andorra model [1]. The second illustrates the idea of local execution which is unique to AKL [7].

*Also in Proceedings of the 2nd Workshop on Extensions to Logic Programming, Springer-Verlag.

[†]Box 1263, S-164 28 KISTA, Sweden; Tel +46-8-752 15 00, Fax +46-8-751 72 30, E-mail {torkel,seif,sverker}@sics.se

2.1 N-Queens Example

The N-queens problem is how to place N queens on an N by N board in such a way that no queen threatens any other. The problem is very well-known, and we will not present any new algorithm. The novelty lies in the way the algorithm is expressed. Let us therefore first familiarise ourselves with some particulars of AKL, especially how the notion of determinism is employed.

In AKL, goals that are *determinate* will be selected first, and goals that are nondeterminate will be delayed. Only when there are no determinate goals will a nondeterminate goal be selected for which alternative clauses are tried. A goal is said to be determinate if there is only one clause that could match the goal. As an example, consider the following two predicates, which are the building blocks in our N-queens program.

```
xcell(1, N, N).  
xcell(0, -, -).  
  
ycell(1, -, -).  
ycell(0, S, S).
```

xcell/3 goals and ycell/3 goals are determinate if the first argument is known, or if the second two arguments are known and different.

In the N-queens problem, the basic constraint is that there may be at most one queen along each row, column, and diagonal. Given that N queens are to be placed on an N by N board, a derived constraint is that there must be exactly one queen along 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 represent each square on the board as a variable V with the possible values 0 (meaning that there is no queen on the square) or 1 (meaning that there is a queen on the square). For a sequence of squares V_1 to V_n , we can now express that *at most one* of these squares is 1 using the xcell/3 goal as follows.

```
?- xcell(V1, N, 1),  
   xcell(V2, N, 2),  
   ...,  
   xcell(Vn, N, n).
```

If more than one V_i is 1, the variable N will be unified with two different numbers, and the goal will fail. Let us simply call the above goal `at_most_one(V1, ..., Vn)`, thus avoiding the overhead of having to write a program to create it. For a sequence of squares V_1 to V_n , we can express that *at least one* of these squares is 1 using the ycell/3 goal as follows.

```
?- S0 = true,  
   ycell(V1, S0, S1),  
   ycell(V2, S1, S2),  
   ...,  
   ycell(Vn, Sn-1, Sn),  
   Sn = false.
```

Only if all the squares are 0, ‘true’ will be unified with ‘false’, and the goal will fail. This goal we call `at_least_one(V1, ..., Vn)`.

An `at_most_one/n` goal will clearly only have solutions where at most one square is given the value 1. This is true in AKL as well as in Prolog, but in AKL it can do more than that. If one of the V_i is given the value 1, its associated `xcell/3` goal becomes determinate, and can therefore be reduced. When it is reduced, N is given the value i , and the other `xcell/3` goals become determinate, and can be reduced. When they are reduced, their first arguments are given the value 0.

Similarly, if in an `at_least_one/n` goal, all variables but one are given the value 0, then their associated `ycell/3` goals become determinate. When they are reduced, their second and third arguments are unified. When only one `ycell/3` goal remains, its second argument will be ‘true’ and its third argument will be ‘false’, and it will therefore be determinate. When it is reduced, its first argument will be given the value 1.

Thus, not only will these goals avoid the undesirable cases, but they will also detect cases where information can be propagated. When no goal is determinate, and therefore no information can be propagated, alternative assignments for variables will be tried by trying alternative clauses for the `xcell/3` and `ycell/3` goals.

A solution to the N -queens problem can now be expressed as follows. For each column, row, and diagonal, consisting of a sequence of variables V_1, \dots, V_n , the goal `at_most_one(V1, ..., Vn)` has to be satisfied. For each column and row, the goal `at_least_one(V1, ..., Vn)` has to be satisfied.

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

$$\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 0 & V_{23} & V_{24} \\ 0 & V_{32} & 0 & V_{34} \\ 0 & V_{42} & V_{43} & 0 \end{array}$$

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 unbound, and all the goals for solving the queens problem 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` goal. By giving one variable the value 1, the others will be given the value 0.

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.

$$\begin{array}{cccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ V_{31} & 0 & 0 & 0 \\ V_{41} & 0 & V_{43} & 0 \end{array}$$

Here we examine the propagation that this state will give rise to. Notice that in row 3, all variables but V_{31} have been given the value 0. This triggers the `at_least_one`

goal 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 not only a solution, but also a fairly good solution. 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 [11]. A very good solution can be obtained if all the $xcell/3$ and $ycell/3$ goals are ordered so that those governing variables closer to the center of the board come before those governing variables further out. If at some step alternative clauses have to be tried for a goal, values will be guessed for variables at the center first. This happens to be a good heuristic for the N-queens problem, even better than the first fail principle which is usually employed.

In this section we have not really described how AKL provides the determinate before nondeterminate functionality, nor how it defines the notion of a clause matching a goal. These topics are unfortunately outside the scope of this paper; the details can be found in [7].

2.2 Sublist Example

In this example, also used in [7], we illustrate how the notion of local execution in AKL makes it possible to write more elegant search-programs. It is a program that finds common sublists of two lists. A Prolog program for the same task follows.

```
sublist([], []).
sublist([X|L], [X|L1]) :- sublist(L, L1).
sublist(L, [X|L1]) :- sublist(L, L1).

?- 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. However, the second goal could be evaluated in advance, and this work could then be reused for the solutions of the first. But, this is not a good solution. The second goal has 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 (and we will now use the “:” construct which is new to AKL).

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

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

The “:” construct, pronounced *wait*, is a conjunction operator, but it has the additional operational meaning that the goals preceding it should be executed locally. What this means will be shown below by illustrating a computation using some of

the apparatus introduced in Section 3. It will be helpful to examine that and following sections in advance, but a sufficient amount of preliminaries are introduced here.

We will use *and-boxes* of the form $\mathbf{and}(G_1, \dots, G_n)_V$, which are conjunctive goals with local variables V , *choice-boxes* of the form $\mathbf{choice}(G_1, \dots, G_n)$, which are disjunctive goals holding different clauses for a goal, and *or-boxes* $\mathbf{or}(G_1, \dots, G_n)$ holding alternative goals that are the result of trying alternative clauses.

Let us start with the initial goal

$\mathbf{and}(\text{sublist}(L, [c, a, t, s]), \text{sublist}(L, [l, a, s, t]))$

The sublist goals are first unfolded into choice-boxes. Note that the first clause disappears because of failure.

$\mathbf{and}(\mathbf{choice}(\mathbf{and}(L = [E|L1], \text{suffix}([E|Z], [c, a, t, s]))_{\{E, L1, Z\}} : \text{sublist}(L1, Z)),$
 $\mathbf{choice}(\mathbf{and}(L = [E|L1], \text{suffix}([E|Z], [l, a, s, t]))_{\{E, L1, Z\}} : \text{sublist}(L1, Z)))$

The and-boxes preceding the wait operators “:” will now execute locally, finding all solutions. Without getting into the details of the computation model it should be intuitive that after a while the following configuration is reached.

$\mathbf{and}(\mathbf{choice}(\mathbf{and}(L = []): \text{true},$
 $\mathbf{and}(L = [c|L1])_{\{L1\}} : \text{sublist}(L1, [a, t, s]),$
 $\mathbf{and}(L = [a|L1])_{\{L1\}} : \text{sublist}(L1, [t, s]),$
 $\mathbf{and}(L = [t|L1])_{\{L1\}} : \text{sublist}(L1, [s]),$
 $\mathbf{and}(L = [s|L1])_{\{L1\}} : \text{sublist}(L1, [])),$
 $\mathbf{choice}(\mathbf{and}(L = []): \text{true},$
 $\mathbf{and}(L = [l|L1])_{\{L1\}} : \text{sublist}(L1, [a, s, t]),$
 $\mathbf{and}(L = [a|L1])_{\{L1\}} : \text{sublist}(L1, [s, t]),$
 $\mathbf{and}(L = [s|L1])_{\{L1\}} : \text{sublist}(L1, [t]),$
 $\mathbf{and}(L = [t|L1])_{\{L1\}} : \text{sublist}(L1, [])))$

The solutions that have been found for the suffix/2 goal give rise to different alternatives in the choice-box. At this stage, different alternatives have to be tried for the choice-box. This will eventually lead to a Cartesian product of the alternatives in the two choice-boxes. When the combinations that fail have been removed, the new configuration is as follows.

$\mathbf{or}(\mathbf{and}(L = []),$
 $\mathbf{and}(L = [a|L1], \text{sublist}(L1, [t, s]), \text{sublist}(L1, [s, t])),$
 $\mathbf{and}(L = [t|L1], \text{sublist}(L1, [s]), \text{sublist}(L1, [])),$
 $\mathbf{and}(L = [s|L1], \text{sublist}(L1, []), \text{sublist}(L1, [t])))$

Execution will now continue within each of the and-boxes, but we will stop here, hopefully having illustrated the point that AKL can express the notion of local execution in a potentially useful manner.

3 The Syntax of Clauses and Goals

A *program atom* is an atomic formula of the form $p(x_1, \dots, x_n)$ with different variables x_1, \dots, x_n , called *head parameters*. A *constraint atom* is any atomic formula

without function symbols in some constraint language. More about constraints in Section 4. It is assumed that these two classes of atomic formulas are disjoint. The remaining syntactic categories pertaining to programs are the following.

$$\begin{aligned}
\langle \textit{guarded clause} \rangle &::= \langle \textit{head} \rangle \textit{:} \langle \textit{guard} \rangle \langle \textit{guard operator} \rangle \langle \textit{body} \rangle \\
\langle \textit{head} \rangle &::= \langle \textit{program atom} \rangle \\
\langle \textit{guard} \rangle &::= \langle \textit{non-empty sequence of atoms} \rangle \\
\langle \textit{atom} \rangle &::= \langle \textit{program atom} \rangle \mid \langle \textit{constraint atom} \rangle \\
\langle \textit{body} \rangle &::= \langle \textit{sequence of atoms} \rangle \\
\langle \textit{guard operator} \rangle &::= \textit{'.'} \mid \textit{'!'} \mid \textit{'|'} \quad (\textit{wait, cut, commit})
\end{aligned}$$

A *definition* is a finite sequence of guarded clauses with the same head atom and the same guard operator, defining the predicate of the head atom. We will speak of wait-definitions, cut-definitions, and commit-definitions. Cut and commit are the *pruning* guard operators. A *program* is a finite set of definitions where no two definitions define the same predicate, and the predicate of every program atom occurring in a clause in the program has a definition in the program. The *local parameters* of a clause are those variables that are not head parameters.

‘%’ below stands for any of the guard operators.

The execution of an Andorra program will be represented as a series of rewrites of goal expressions. The syntax of these expressions is defined as follows.

$$\begin{aligned}
\langle \textit{and-box} \rangle &::= \mathbf{and}(\langle \textit{non-empty sequence of local goals} \rangle)_{\langle \textit{set of variables} \rangle} \\
\langle \textit{local goal} \rangle &::= \langle \textit{atom} \rangle \mid \langle \textit{choice-box} \rangle \\
\langle \textit{choice-box} \rangle &::= \mathbf{choice}(\langle \textit{sequence of guarded goals} \rangle) \\
\langle \textit{guarded goal} \rangle &::= \langle \textit{configuration} \rangle \langle \textit{guard operator} \rangle \langle \textit{seq. of atoms} \rangle \\
\langle \textit{or-box} \rangle &::= \mathbf{or}(\langle \textit{sequence of configurations} \rangle) \\
\langle \textit{configuration} \rangle &::= \langle \textit{and-box} \rangle \mid \langle \textit{or-box} \rangle \\
\langle \textit{open goal} \rangle &::= \langle \textit{configuration} \rangle \mid \langle \textit{local goal} \rangle \\
\langle \textit{goal} \rangle &::= \langle \textit{open goal} \rangle \mid \langle \textit{guarded goal} \rangle
\end{aligned}$$

The guarded goals in a choice-box have the same guard operator.

In the following, the letters R , S , and T stand for sequences of goals, A for a sequence of atoms, and C for a sequence of constraint atoms. Any of these may stand for the empty sequence. The letter G (occasionally X) will be used for goals, and for the guard in a guarded clause. Concatenation of sequences will be written using comma, which is not likely to be confused with the use of comma for separating arguments.

The symbol **fail** will be used to denote the empty or-box and the empty choice-box, regarded as collapsing to the same object. G is a *total failure* if G is either **fail** or an or-box $\mathbf{or}(G_1, \dots, G_n)$ where each G_i is a total failure.

The set of variables attached to an and-box contains the variables *local* to that box, introduced in local forking or in defining an initial goal, as described below. An and-box of the form $\mathbf{and}(C)_V$ will normally be written as C_V .

A variable in G is *external* to a subgoal G' of G if it is local to some and-box properly containing G' , or if it is not local to any and-box.

In a guarded goal, the configuration preceding the guard operator is the *guard* of the goal. The atoms following the guard operator are not regarded as goals, but become such when one of the promotion rules defined in Section 5 is applied to the goal.

We will speak of *subgoals* of goals in the obvious sense. That is,

1. G is a subgoal of G ,
2. Subgoals of G_1, \dots, G_n are subgoals of $\mathbf{and}(G_1, \dots, G_n)_V$, $\mathbf{or}(G_1, \dots, G_n)$, and $\mathbf{choice}(G_1, \dots, G_n)$,
3. Subgoals of G are subgoals of $G \% A$.

Subgoals of G other than G itself are called *proper* subgoals.

Usually, we will be talking about *occurrences* of subgoals, rather than of just the syntactic form of a subgoal. So the definitions below should in most cases be read as dealing with occurrences of goals. Since confusion between form and occurrence is unlikely, the distinction will mostly be implicit.

In the case where G is an and-box, an or-box, or a choice-box, we will speak of subgoals of G (including G itself) as occurring *in* G , and of G as a *surrounding* box. Again the qualification “properly” excludes G itself.

We will need the notion of *or-component*. A subgoal G of an or-box $G' = \mathbf{or}(G_1, \dots, G_n)$ is an or-component of G' if G is G_i or is an or-component of G_i for some $i = 1, \dots, n$. For convenience we will also regard a goal of the form C_V as an (improper) or-component of itself.

Finally, the notion of *ordered context*. An and-box G in G' occurs in an ordered context if it is a subgoal of some cut-guarded choice-box G'' in G' , but not of any commit-guarded choice-box in G'' . More informally, the nearest surrounding pruning choice is a cut. Although used in the formulation of the non-deterministic promotion rule, this notion will otherwise play no role in the present paper.

4 Constraints

Constraints will be regarded as formulas in some constraint language. The only constraint formulas that occur in goals are atomic constraints without function symbols. The letters σ , τ , and θ will be used for conjunctions of such constraints.

In the following, $\exists\sigma$ stands for the existential closure of σ , and $\exists V$, where V is the set $\{x_1, \dots, x_n\}$, for (any permutation of) the quantifier sequence $\exists x_1 \cdots \exists x_n$. We allow V to be empty, in which case $\exists V$ is mere decoration. Overlined letters \bar{x} , \bar{y} , and \bar{z} will be used for sequences x_1, \dots, x_n of variables when their length n (≥ 0) is immaterial. When the notation $G(\bar{x}, \bar{y})$ or $\sigma(\bar{z})$ etc is used, it is not supposed that all the variables in \bar{x} , \bar{y} , or \bar{z} actually occur in G or σ , but only that no other variables occur in G or σ .

We assume given some complete and consistent constraint theory **TC** defining the following logical properties of constraints:

1. σ is *satisfiable* iff $\mathbf{TC} \models \exists\sigma$,
2. θ is *unconstrained by* σ (or: σ *does not restrict* θ) *outside* V iff

$$\mathbf{TC} \models \theta \supset \exists V(\sigma \wedge \theta)$$

This latter condition will also be expressed by saying that σ does not restrict θ with respect to the complement of V .

The symbol **true** is used to denote a variable-free atomic formula for which it holds that $\mathbf{TC} \models \mathbf{true}$. It will also be assumed that there is a variable-free atomic formula **false** for which $\mathbf{TC} \models \neg\mathbf{false}$. No further assumptions concerning the properties of \mathbf{TC} will be made unless explicitly stated.

That \mathbf{TC} is assumed to be complete is just a matter of theoretical convenience. Function symbols are excluded from constraint atoms because they have no role to play in the interpretation or execution of a program. This is no restriction from a theoretical point of view, since any definitions of the constraint atoms may be part of \mathbf{TC} . In specifying a particular language following the AKL model we need to define not only the constraint theory \mathbf{TC} but also the class of constraint atoms. In the familiar cases of Herbrand terms (or finite trees) and rational trees, we will speak of “AKL with Herbrand equalities” and “AKL with rational tree equalities”, meaning that the predicate p of a constraint atom either is equality, or else has a definition in the constraint theory of the form

$$\forall x(p(\bar{x}) \equiv s = t)$$

where s and t are terms containing no variable not in \bar{x} . The constraint theories are in these cases \mathbf{HC} , the complete elementary theory of Herbrand terms, and \mathbf{RC} , the complete elementary theory of rational trees, both extended with the above definitions.

For a sequence R of goals, $\sigma(R)$ denotes the conjunction of the constraint atoms in the sequence, or true if there are no constraint atoms in R . The *environment* of (an occurrence of) a subgoal of a configuration G , or more explicitly its environment in G , is the conjunction of $\sigma(R)$ for every properly surrounding and-box $\mathbf{and}(R)_V$ in G , or **true** if there is no such and-box.

Constraints σ and τ are *incompatible* if $\sigma \wedge \tau$ is unsatisfiable. An occurrence of C_V in G is *quiet* if $\sigma(C)$ does not restrict the environment of C_V outside V .

5 Rewrite Rules

The rewrite rules below are to be understood as applicable to any subgoal of a goal. That is, we define the rewrite relation on goals

$$G \rightarrow G'$$

as meaning that G' is obtainable from G by rewriting one occurrence of some subgoal X of G as X' in accordance with one of the rewrite rules

$$X \Rightarrow X'$$

defined below. (We will say that $G \rightarrow G'$ via $X \Rightarrow X'$.) So references to the environment of X refer to the environment of X as a subgoal of G . In speaking of the goal to which a rule is applied, we will always mean X rather than G ; an application of a rule to a subgoal of G is an application within G . A deterministic operation is an application of any rule other than non-deterministic promotion.

Note that X and X' are always open goals, and X is never an or-box.

Local Forking

A program atom subgoal A is rewritten by

$$A \Rightarrow \mathbf{choice}(\mathbf{and}(G_1)_{V_1} \% A_1, \dots, \mathbf{and}(G_n)_{V_n} \% A_n)$$

where $H :- G_i \% A_i$ ($i = 1, \dots, n$) is the sequence of clauses defining the predicate of A , with the arguments of A substituted for the head parameters, and the local parameters of the i :th clause replaced by the variables in the set V_i . When G is rewritten to G' by applying local forking to an atomic subgoal of G , the sets V_i are chosen to be disjoint from the set of variables in G .

Failure Propagation

$$\mathbf{and}(R, \mathbf{fail}, S)_V \Rightarrow \mathbf{fail}$$

Choice Elimination

$$\mathbf{choice}(R, \mathbf{fail} \% A, S) \Rightarrow \mathbf{choice}(R, S)$$

Environment Synchronization

$$\mathbf{and}(R)_V \Rightarrow \mathbf{fail}$$

if $\sigma(R)$ is incompatible with the environment of the and-box.

Guard Distribution

$$\mathbf{choice}(R, \mathbf{or}(G, S) \% A, T) \Rightarrow \mathbf{choice}(R, G \% A, \mathbf{or}(S) \% A, T)$$

Deterministic Promotion

$$\mathbf{and}(R, \mathbf{choice}(C_V \% A), S)_W \Rightarrow \mathbf{and}(R, C, A, S)_{V \cup W}$$

if C_V is satisfiable, and in case $\%$ is cut or commit is also quiet.

Cut Promotion

$$\mathbf{choice}(R, C_V ! A, S) \Rightarrow \mathbf{choice}(R, C_V ! A)$$

if S is non-empty and C_V is satisfiable and quiet.

Commit Promotion

$$\mathbf{choice}(R, C_V | A, S) \Rightarrow \mathbf{choice}(C_V | A)$$

if R or S is non-empty and C_V is satisfiable and quiet.

Non-Deterministic Promotion

$$\begin{aligned} &\mathbf{and}(T_1, \mathbf{choice}(R, C_V : A, S), T_2)_W \Rightarrow \\ &\quad \mathbf{or}(\mathbf{and}(T_1, C, A, T_2)_{V \cup W}, \mathbf{and}(T_1, \mathbf{choice}(R, S), T_2)_W) \end{aligned}$$

if R or S is non-empty. There are two further conditions associated with this rule. First, if the and-box occurs in an ordered context, it is required that R is empty. This stipulation is necessary for the concept of “the first solution” of a cut-guard to be well-defined. If it is not included, we get the rule for unordered hard cut. In this paper, the distinction between the ordered and the unordered hard cut will play no role, and the concept of ordered context will not be used in the following. The second condition for non-deterministic promotion to be applicable is a bit complicated and merits separate treatment.

6 Stability and Non-Deterministic Promotion

A couple of definitions are useful in discussing non-deterministic promotion: a cut-guard or commit-guard in G of the form C_V is *suspended* if it is not quiet. A step *unfreezes* a suspended guard if the guard is quiet after that step. An application of environment synchronization, commit promotion, or cut promotion is *non-trivial* if the environment of the goal to which the operation is applied is consistent. Other operations are always non-trivial.

In an application of non-deterministic promotion as exhibited above we will say that the rule is applied *with respect* to the guard C_V . Also, an and-box of the form

$$\mathbf{and}(T_1, \mathbf{choice}(R, C_V : A, S), T_2)_W$$

where R or S is non-empty will be called a *candidate* for non-deterministic promotion.

The basic Andorra principle of performing deterministic promotion in preference to non-deterministic promotion translates into a necessary condition for applying non-deterministic promotion, viz. that no deterministic operation is applicable to or within the and-box. This will be called the non-determinacy condition.

In AKL, the basic Andorra principle is given a stronger interpretation: non-deterministic promotion is to be delayed until (i) the non-determinacy condition is satisfied, and (ii) non-trivial deterministic operations on or within the and-box cannot become possible as a result of any future changes in the environment of the and-box through deterministic operations. This will be called the AKL principle.

The AKL principle will be seen to be a consequence of the restrictions on non-deterministic promotion now to be formulated. We distinguish between the AKL principle and the particular restrictions to be described here, since these restrictions, although guided by the AKL principle, also incorporate specific ideas concerning what is useful in programming. We need the concept of *stability*: an and-box G which is a subgoal of an and-box G' is stable (relative to G') if

- i) no deterministic operation is applicable to or within G , and
- ii) no series of rewrites of G' , in which no restriction is imposed on non-deterministic promotion apart from the non-determinacy condition (and, for ordered

hard cut, the restriction on ordered contexts), can lead to a goal in which a non-trivial deterministic operation is applicable to or within G .

In speaking of stable and-boxes below, we will mean boxes stable relative to the whole configuration being rewritten. Note that the property of stability is well-defined (although in general undecidable) since this definition does not invoke any further restrictions on non-deterministic promotion.

The conditions on non-deterministic promotion can now be stated: non-deterministic promotion is applicable to an and-box

$$G = \mathbf{and}(T_1, \mathbf{choice}(R, C_V : A, S), T_2)_W$$

if and only if G has a nearest surrounding stable and-box G' such that every candidate for non-deterministic promotion in G' has G' as its nearest surrounding stable and-box.

Given these restrictions, it easily follows that the AKL principle will be satisfied. For no deterministic operation is applicable to or within a subgoal of a stable and-box, and if G is a subgoal of a stable and-box, further rewrites may enable non-trivial deterministic operations within G only as a result of applications of non-deterministic promotion to surrounding and-boxes.

Additional conditions may be imposed on non-deterministic promotion, stipulating e. g. that non-deterministic promotion must be applied with respect to the left-most possible solved guard, or that it must be applied to an innermost eligible candidate. However, such further restrictions need not be considered in this presentation, since the basic restriction alone implies that the Andorra principle holds, and the completeness theorem to be formulated below holds whatever such further restrictions are imposed.

In an implementation of AKL one will need to replace the abstract definition of stability by a more manageable sufficient condition for stability. Of course the part of the definition stipulating that no deterministic operation is applicable to G is non-problematic. What we need to consider are sufficient conditions for ruling out any future non-trivial application of the rules for environment synchronization, commit promotion, and cut promotion.

To deal with environment synchronization, we can give a condition that is independent of the constraint theory TC. It is easily shown that any future non-trivial environment synchronization in G can be ruled out if the following *environment condition* holds:

For every and-box $\mathbf{and}(R)_V$ in G with environment τ in G , $\sigma(R) \wedge \tau$ does not restrict the environment of G with respect to the variables external to G .

We also need to formulate a *suspension stability condition*, that is, a condition that will ensure that no future deterministic step will unfreeze any suspended guard in G , unless the environment of G becomes inconsistent. The environment condition is not in general itself a suspension stability condition. For a counterexample, suppose G has external variables y_1 and y_2 , on which the environment of G imposes no condition, and a suspended constraint $x < y_1$, where the environment of the constraint in G is $x < y_2$. Assuming $<$ to refer to real numbers, the conjunction of these does not restrict the environment of G , but if $y_2 < y_1$ is added to

that environment, $x < y_1$ will become quiet. Similar counterexamples apply if the constraints include equalities in the field of real numbers, or inequalities between Herbrand terms. These examples suggest that a suspension stability condition that covers many different constraint theories must be rather stringent. One sufficient condition is clearly that there are in G no suspended constraints containing external variables.

For the purposes of the present paper, it is not necessary to make any particular stipulations regarding suspension stability conditions. The soundness proof in fact makes no use at all of the stability condition, and for the completeness proof the suspension stability condition is irrelevant.

In the special cases of Herbrand equalities and rational tree equalities it turns out that the environment condition is in fact itself a suspension stability condition. These two types of constraints seem to be the only naturally occurring ones for which this holds.

7 Some Basic Properties of the Rules

The first thing to note, by an inspection of the rules and of the syntax of goals, is that rewriting a goal in accordance with one of the rules does in fact result in another goal. Thus, for example, replacing an occurrence of an and-box in a goal by an occurrence of an or-box yields a new goal.

An *initial* goal has the form $\mathbf{and}(A)_V$, with V a subset of the set of variables occurring in any of the atoms in the non-empty sequence A (the existentially quantified variables in the initial goal). We define a *computation* as a finite or infinite sequence G_1, G_2, \dots of goals, where G_1 is an initial goal and $G_i \rightarrow G_{i+1}$ for every non-final i . A computation G'_1, G'_2, \dots is a *subcomputation* of G_1, G_2, \dots if each G'_i is a subgoal of G_{k_i} , for some subsequence G_{k_1}, G_{k_2}, \dots of G_1, G_2, \dots . We will use the notation $G \rightarrow^* G'$ to mean that there is a computation in which $G = G_i$ and $G' = G_j$ for some $i < j$.

It is easily seen that all goals occurring in a computation are and-boxes or or-boxes, that is, configurations. In particular, all configurations after the first top-level application of non-deterministic promotion (if there is such) will be or-boxes, in general nested, with no flattening taking place, since there is no rule for flattening or-boxes. Or-boxes occurring in guards, on the other hand, may be flattened in the course of a computation. (By using the guard distribution rule in combination with the choice elimination rule we can also reduce $\mathbf{or}(G)$ to G and $\mathbf{or}(R, \mathbf{fail})$ to $\mathbf{or}(R)$.)

We'll adopt Prolog terminology and speak of an initial goal $\mathbf{and}(A)_V$ as a *query*. We will sometimes write the query as $\exists VA$, or just as A if V is empty. C_W is an *answer* to a query G (relative to a program) if C_W is an or-component of some G' for which $G \rightarrow^* G'$. Note that $\sigma(C)$ need not be satisfiable: if an answer $\sigma(C)$ is satisfiable, it is a *solution*. A computation *fails* if it ends with a total failure—inspection of the rules shows that no rewrite can be made of a total failure. A query G fails if at least one computation starting with G fails, it *succeeds* if it has at least one solution.

A *complete* computation is one that is either finite, with no rule applicable to the final goal, or is infinite and satisfies the condition that no subgoal appears in the course of the computation such that

- i) the subgoal remains unchanged throughout the computation, and
- ii) for infinitely many configurations in the computation, some rule is applicable to the subgoal.

In other words: in a complete computation no stone is left forever unturned, but every subgoal that appears is either eventually deleted or transformed through an application of some rule to that or some other subgoal, or else after a certain point no rule ever becomes applicable to the subgoal.

Computations in which every guard computation terminates, which will be referred to as *normal* computations, play an important role in AKL. In formal terms, a normal computation is one in which there is no infinite subcomputation beginning with a guard, and no choice-box to which guard distribution is applied infinitely many times in the computation. The use of deep guards in AKL implies that non-terminating guard computations are possible, but theoretically as well as from the point of view of actual programming they are usually to be regarded as pathological.

8 AKL as a Logic Programming Language

Among the styles of programming supported by AKL is traditional declarative logic programming, and it is accordingly of interest to consider the traditional issues of soundness and completeness. For proofs of the theorems stated below, see [4].

The soundness result to be formulated refers to the completion of a program, in a fairly obvious sense. That is, for each definition

$$p(\bar{x}) :- G_i \% A_i \quad (i = 1, \dots, k)$$

in the program in which $\%$ is not (hard or soft) cut, we form the logical formula

$$\forall \bar{x}(p(\bar{x}) \equiv \exists V_1(G_1 \& A_1) \vee \dots \vee \exists V_k(G_k \& A_k))$$

where V_i is the set of local parameters of the i :th clause. Here and in the following we use the convention that sequences of atomic formulas, when they occur in formulas as above, stand for the conjunction of the formulas in the sequence, or for **true** if the sequence is empty. For cut definitions, we form instead the formula

$$\begin{aligned} \forall \bar{x}(p(\bar{x}) \equiv & \exists V_1(G_1 \& A_1) \vee \\ & \neg \exists V_1 G_1 \& \exists V_2(G_2 \& A_2) \vee \\ & \dots \vee \\ & \neg \exists V_1 G_1 \& \dots \& \neg \exists V_{k-1} G_{k-1} \& \exists V_k(G_k \& A_k) \end{aligned}$$

We call the resulting set of formulas Σ^* .

Talk of soundness, correctness, and logical consequence will in the following refer to the theory $\mathbf{T} = \mathbf{TC} \cup \Sigma^*$. Recall that \mathbf{T} may in general be inconsistent, in which case soundness with respect to \mathbf{T} is not a property of great interest. It is occasionally of interest, for technical reasons, to investigate sufficient conditions for \mathbf{T} to be consistent. However, that \mathbf{T} is consistent does not guarantee that computed answers are sound in any interesting (not merely technical) sense. Rather, the soundness result to be presented and the terminology used are predicated on

the assumption that the axioms of \mathbf{T} are in fact true on the intended interpretation of the program. Preoccupation with soundness in this sense seems reasonable since this assumption is often justified in declarative-style logic programming.

Every goal has a natural interpretation as a logical formula. We define an operation $*$ assigning a formula G^* to any goal G :

1. For an atomic goal A , A^* is A .
2. $(\mathbf{fail} \% A)^*$ is \perp
3. \mathbf{fail}^* is \perp
4. For guard operators other than hard cut, $(\mathbf{or}(G_1, \dots, G_n) \% A)^*$ is

$$(G_1 \% A)^* \vee \dots \vee (G_n \% A)^*$$

5. For hard cut, $(\mathbf{or}(G_1, \dots, G_n) ! A)^*$ is

$$(G_1 ! A)^* \vee (\neg G_1^* \& (G_2 ! A)^*) \vee \dots \vee (\neg G_1^* \& \dots \& \neg G_{n-1}^* \& (G_n ! A)^*)$$

6. $\mathbf{or}(G_1, \dots, G_n)^*$ is $G_1^* \vee \dots \vee G_n^*$.

7. $(\mathbf{and}(G_1, \dots, G_n)_V \% A)^*$ is $\exists V(G_1^* \& \dots \& G_n^* \& A)$

8. $\mathbf{and}(G_1, \dots, G_n)_V$ is $\exists V(G_1^* \& \dots \& G_n^*)$

9. For wait and commit choices, $\mathbf{choice}(G_1, \dots, G_n)^*$ is $G_1^* \vee \dots \vee G_n^*$.

10. For cut choices, $\mathbf{choice}(G_1 \% A_1, \dots, G_n \% A_n)^*$ is

$$(G_1 \% A_1)^* \vee (\neg G_1^* \& (G_2 \% A_2)^*) \vee \dots \vee (\neg G_1^* \& \dots \& \neg G_{n-1}^* \& (G_n \% A_n)^*)$$

A computation $G_1 \rightarrow G_2 \rightarrow G_3 \dots$ is *logical* if it holds that $\mathbf{T} \models G_i^* \equiv G_{i+1}^*$ for every non-final i . A program is *logical* if every computation from that program is logical. Clearly all answers and failures obtained in a logical computation are *sound*, in the sense that the following two principles hold:

Soundness of answers: if C_W is an answer to $\exists V A$ in a logical computation,

$$\mathbf{T} \models \sigma(C) \supset A$$

Soundness of failure: if $\exists V A$ fails in a logical computation,

$$\mathbf{T} \models \neg A$$

In general, an AKL computation may or may not be logical. The soundness theorem stated below gives sufficient (and in practice “almost” necessary) conditions for a program to be logical. For an explanation and discussion of the concepts defined below, see [4].

A cut-guarded definition has *indifferent* guards if for every clause

$$p(\bar{x}) :- G(\bar{x}, \bar{y}) ! B(\bar{x}, \bar{y})$$

in the definition, where the variables in \bar{y} are the local parameters of the clause, it holds that

$$\mathbf{T} \models G(\bar{x}, \bar{y}) \ \& \ G(\bar{x}, \bar{z}) \supset (\exists \bar{w} B(\bar{x}, \bar{y}) \equiv \exists \bar{w} B(\bar{x}, \bar{z}))$$

where the sequence \bar{w} contains those variables in \bar{y} that do not occur in $G(\bar{x}, \bar{y})$.

A simple syntactic sufficient condition for the guards of a cut definition to be indifferent is that they are *isolated*, in the sense that that no local parameter in any clause in the definition occurs both in the guard and in the body. (Note that this comfortably covers negation as failure.)

A commit definition has *authoritative* guards if for every clause

$$p(\bar{x}) \text{ :- } G(\bar{x}, \bar{y}) \mid B(\bar{x}, \bar{y})$$

in the definition it holds that

$$\mathbf{T} \models G(\bar{x}, \bar{y}) \supset (p(\bar{x}) \equiv \exists \bar{w} B(\bar{x}, \bar{y}))$$

where the sequence \bar{w} contains those variables in \bar{y} that do not occur in $G(\bar{x}, \bar{y})$.

With these definitions, we have the following

Soundness theorem: All programs satisfying the following conditions are logical:

- i) All cut definitions have indifferent guards.
- ii) All commit definitions have authoritative guards.

(For weak cut or commit these conditions can be weakened; see [4].) While the quietness restrictions on pruning promotion operations are essential for the soundness theorem to hold, the restrictions on non-deterministic promotion are irrelevant to the proof, as is the quietness condition in the case of deterministic promotion of a pruning guard.

As far as completeness is concerned, a result analogous to that for SLD resolution applies. The completeness theorem given here deals only with wait-clauses, i.e. the part of AKL corresponding to the execution of Horn clauses in Prolog. The constraint theory, however, is arbitrary. So, given a program composed entirely of wait-definitions, let Σ be the set of corresponding definite clauses:

$$\forall \bar{x} \forall V_i (G_i \ \& \ A_i \supset p(\bar{x}))$$

It is easily verified that $\Sigma \models G_{i+1}^* \supset G_i^*$ for all configurations in a computation $\dots \rightarrow G_i \rightarrow G_{i+1} \rightarrow \dots$. Hence AKL execution is sound with respect to Σ , in the following sense: if $C(\bar{x}, \bar{y})_W$ is a solution to the query $\mathbf{and}(A(\bar{x}))_V$, it holds that (i) $\Sigma \models C(\bar{x}, \bar{y}) \supset A(\bar{x})$, and (ii) $\mathbf{TC} \models \exists \bar{x} \exists \bar{y} C(\bar{x}, \bar{y})$. (Here we use the observation that if C_W is an answer to $\exists V A$, $C_{W \setminus V}$ is an answer to A .) The following restricted converse holds:

Completeness theorem: Suppose Π is a complete normal computation starting with the query

$$\mathbf{and}(A(\bar{x}))_V$$

Then if (i) and (ii) hold, Π contains (i.e. some configuration in Π has as an or-component) a solution $D(\bar{x}, \bar{z})_W$ such that $\models C(\bar{x}, \bar{y}) \supset \exists \bar{z} D(\bar{x}, \bar{z})$.

The restriction to normal computations is needed because of the conditions for non-deterministic promotion to be applicable in AKL. A goal of the form **and**(p, q) will loop if p is non-deterministic and q is deterministic and loops, even if p has **false** as the body of each clause. When such a goal occurs in a guard it may prevent a solution given by another clause from being promoted. So the theorem above states that essentially this is the only problematic situation.

The conditions of the theorem are clearly satisfied for *any* complete computation if there is no recursion involved in the guard of any clause in the program, i.e. the guards are essentially flat. With recursive guards, there may or may not be any complete computation with terminating guard computations, and for a given logical consequence there may or may not be any computation in which that consequence is found.

9 Discussion

AKL brings together many ideas from different camps and different people in a quest to combine Prolog, committed-choice languages, and constraint logic programming in a single unified framework.

D. H. D. Warren proposed the basic Andorra model (first described in [5]) as a way to combine or-parallelism with dependent and-parallelism in the execution of pure definite clauses. The basic Andorra model has been implemented, exploiting or-parallelism in combination with dependent and-parallelism in the execution of Prolog programs [3].

The potential of the basic Andorra model as a basis for combining Prolog and committed choice languages was first advocated by Haridi and Brand [5]. Independently, Bahgat and Gregory generalised the basic Andorra model by allowing full Parlog execution during the deterministic phase in the language Pandora [1].

A set of rewrite rules on AND/OR-trees that potentially unifies the abilities of Prolog and GHC was developed as a joint effort between Warren and us. The rewrite rules themselves provide no control, but the Andorra idea of giving priority to deterministic computation is the foundation for the different control principles proposed.

Warren, who calls these rules the Extended Andorra Model, has proposed an implicit control regime for these rules with the goal to provide good behaviour for Prolog programs using the annotations `cut`, `commit` and `sequential conjunction`.

We have earlier developed a formal computation model derived from these rules for the language framework Kernel Andorra Prolog (KAP) [6]. AKL is a refined instance of KAP. For more details on this relation see [7]. The work of Vijay Saraswat was very influential [9], in particular for the notions of constraints and constraint operations.

AKL is being implemented and the results so far are promising. The current implementation covers everything described in this paper, and more. In our AKL environment, we use public domain code originally written in Prolog (by O’Keefe and others) with minor modifications only. Other Prolog programs have also been easy to adapt. Committed-choice examples run with no modifications except simple changes to the syntax. The AKL prototype is available for research purposes.

Acknowledgements

We would like to thank Johan Montelius of the Andorra group at SICS, and the members of ESPRIT Project 2471 (PEPMA), for their contributions to this work.

References

- [1] Reem Bahgat and Steve Gregory. Pandora: Non-deterministic parallel logic programming. In *Proceedings of ICLP'89*. MIT Press, 1989.
- [2] Francisco Bueno and Manuel Hermenegildo. An automatic translation scheme from Prolog to the Andorra Kernel Language. PEPMA Internal Report, March 1991.
- [3] Vitor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the Basic Andorra model. In *Proceedings of ICLP'91*. MIT Press, 1991.
- [4] Torkel Franzén. Logical aspects of the Andorra Kernel Language. SICS Research Report R91:12, Swedish Institute of Computer Science, 1991.
- [5] Seif Haridi and Per Brand. Andorra Prolog, an integration of Prolog and committed choice languages. In *Proceedings of FGCS'88*, 1988.
- [6] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its computation model. In *Proceedings of ICLP'90*. MIT Press, 1990.
- [7] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *Proceedings of the 1991 International Logic Programming Symposium*. MIT Press, 1991. (Revised version of SICS Research Report R91:08, to be published).
- [8] Michael J. Maher. Logic semantics for a class of committed choice programs. In *Proceedings of the Fourth International Conference on Logic Programming*. MIT Press, 1987.
- [9] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1990.
- [10] Kazunori Ueda. Guarded horn clauses. Technical Report TR-103, ICOT, June 1985.
- [11] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
- [12] Rong Yang. Solving simple substitution ciphers in Andorra-I. In *Proceedings of the Sixth International Conference on Logic Programming*. MIT Press, 1989.