ISRN SICS-R--91/8--SE

Programming Paradigms of the Andorra Kernel Language

by
Sverker Janson and Seif Haridi

SICS research report
R91:08
ISSN 0283-3638

SICS/R91:08

Programming Paradigms of the Andorra Kernel Language

Sverker Janson Seif Haridi

April 10, 1991

SICS Research Report R91:08
Swedish Institute of Computer Science
Box 1263, S-164 28 KISTA, Sweden

Abstract

The Andorra Kernel Language (AKL) is introduced. It is shown how AKL provides the programming paradigms of both Prolog and GHC. This is the original goal of the design. However, it has also been possible to provide capabilities beyond that of Prolog and GHC. There are means to structure search, more powerful than plain backtracking. It is possible to encapsulate search in concurrent reactive processes. It is also possible to write a multi-way merger with constant delay. In these respects AKL is quite original. Although AKL is an instance of our previously introduced Kernel Andorra Prolog framework, this exposition contains important extensions, and a considerable amount of unnecessary formal overhead has been stripped away.

1 Introduction

The Andorra Kernel Language (AKL) is a general concurrent logic programming language that is based on an instance of the Kernel Andorra Prolog (KAP) control framework for the Extended Andorra Model [8].

This paper presents a language design. The purpose is to, by means of more or less familiar examples, illustrate the possibilities provided by AKL. This is not to say that implementation has been set aside. We have a prototype implementation of the language, which is showing good results. However, it is felt that there is a need to first explain the properties of the language, before discussing ingenious implementation techniques.

AKL is a concurrent language. Very few aspects of the language are order dependent, which means that there should be ample possibilities for parallel execution. We will touch upon how to provide a potential for parallel execution when writing programs.

The paper is organised as follows. Section 2 provides some background. Section 3 defines AKL and its computation model. Section 4 shows that the programming paradigms of Prolog and GHC are available by providing translations into AKL. It is also shown how to improve upon the translation of "Prolog" in several ways. Section 5 is devoted to nondeterministic computation. First, it is shown how to realise finite domain constraint programming in AKL. Then, it is shown how to produce a Cartesian product effect, combining solutions for two goals. Finally, the Cartesian product is applied to only partially computed solutions, with improved behaviour as the result. Section 6 shows how to make use of an encapsulated nondeterministic computation within a reactive computation to do multi-way merge with constant delay.

2 Background and Related Work

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

The Andorra model was proposed by Warren [14] as a basic tool for combining or-parallelism with general and-parallelism in the execution of pure definite clauses (see section 4.2.3). This idea owes much to the notion of determinacy in P-Prolog [16]. The Andorra model has been implemented providing stream-and parallelism in combination with or-parallelism [3].

The potential of the Andorra model as a basis for combining Prolog and committed-choice languages was first realised by Haridi [7, 6]. Independently, Bahgat and Gregory extended the basic Andorra model by allowing full Parlog execution during the deterministic phase in the language Pandora [1].

As a joint effort between us and Warren, an Extended Andorra Model (EAM) was developed. It is a set of rewrite rules on AND/OR-trees that potentially unifies the abilities of Prolog and GHC. In some sense, the EAM is for AKL what definite clause resolution is for Prolog. The EAM itself provides no control, but the Andorra idea of giving priority to deterministic computation is the foundation for the different control principles proposed for the EAM.

Warren has proposed an implicit control regime for the EAM with the goal to provide good behaviour for programs without extra annotations apart from cut and commit [15].

We have earlier developed a formal computation model based on the EAM for the language framework Kernel Andorra Prolog (KAP) [8]. There we added the wait guard operator to delimit local execution, and the model was based on the notion of constraints and constraint operations. For the latter notions, the work of Vijay Saraswat was very influential [9].

A language that is an instance of the KAP framework uses a subset of the guard-operators and the constraint operations, and it may also further restrict the basic control principles provided. For example, GHC is an instance of KAP using the guard operator commit and (implicitly) the constraint operation tell_0.

AKL is an instance of KAP using all three guard operators proposed for KAP and (implicitly) the constraint operation tell_ω. The control has been restricted to allow quiet pruning only. This instance has been chosen for its nice properties, such as being comparatively insensitive to the order of execution of goals. We have been careful to preserve the reactive aspects of the language as well as providing means for controlling search.

3 The Andorra Kernel Language

In this section we introduce the Andorra Kernel Language (AKL) and its computation model. For a thorough treatment of the AKL rules and their logical properties see [5, 4].

The computation model generalises definite clause resolution where program clauses are resolved against a goal clause. Here, nested goal expressions built from atomic goals, conjunction, and disjunction—AND/OR trees—are worked upon by rewrite rules.

In the following sections the syntax of AKL is defined, then its computation states, and finally we present the formal rewrite rules that form the basis of the computation model.

3.1 The Language

The syntactic categories pertaining to programs follow.

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

A *constraint atom* is any formula in some constraint language (as defined by the constraint system used). In this paper we will only use the constraint system of Prolog and GHC. Thus, constraint atoms are equalities on terms, as usual. A constraint atom of the form $X = t$ or $X = Y$ is called a binding. We assume that

a unification algorithm is available, and that it is used to establish the consistency of a conjunction of equality constraints, and when consistent also reducing it to a conjunction of bindings.

A *program atom* is an atomic formula of the form $p(X_1, \ldots, X_n)$, with different variables $X_1, \ldots, X_n$. It is for technical simplicity that program atoms have distinct variables as arguments. It makes the language description completely independent of the constraint system. Programs with head unification may be translated by putting equalities corresponding to head unification in the guard. Conversely, programs with terms as arguments to body goals may be translated by putting the corresponding equalities together with the goal.

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, depending on the guard operator. Cut and commit are the *pruning* guard operators. The wait-operator only delimits local execution, as discussed below. A *program* is a finite set of definitions, satisfying the condition that the predicate of every program atom occurring in a clause in the program has a definition in the program. The *local variables* of a clause are those variables that are not formal parameters (and thus do not occur in the head).

AKL can support several kinds of aggregate constructs, depending on the constraint system used. Here, we will present and use a derived form of bagof construct, which is written as in Prolog. For technical simplicity, the collected term is restricted to be a variable not occurring elsewhere in the clause, and the goals for which solutions are collected may not be prefixed by existential quantification. Both of these restrictions can be relaxed by the use of simple source-to-source transformations.

We also adopt the syntactic convention that when a guard operator is omitted in a clause, its default position is at the neck of the clause. If another clause in the same definition has an explicit guard operator, it is used, otherwise the default is the wait guard operator.

3.2 Configurations

Configurations are nested expressions built from atoms and components called boxes. The precise logical reading of these expressions is given by the declarative semantics of AKL [5, 4]. The syntax of these expressions is defined as follows.

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

Roughly, the and-boxes and the guarded goals are conjunctions, or-boxes and choice-boxes disjunctions, and bagof-boxes are set-abstractions. The variables with which an and-box is indexed are existentially quantified within the box.

In the following, the letters R, S, and T stand for sequences of goals, A for an atom, B for a sequence of atoms, and C for a sequence of constraint atoms. (A sequence may be empty.) The letter G will be used for goals, and for the sequence of atoms in the guard of a guarded clause. The letters V and W stand for sets of variables. Concatenation of sequences will be written using comma. The symbol **fail** denotes the empty or-box. The symbol '%' stands for a guard operator.

The *constraint* of an and-box is the conjunction of the constraint atoms appearing as members of the and-box. The *environment* of (an occurrence of) a goal in a configuration is the conjunction of all constraints of all its surrounding and-boxes. A variable is *external* to an and-box if it is local to a surrounding and-box. An and-box, $\mathbf{and}(C)_V$, is *solved* if it contains constraint atoms only, in which case it may be written as C_V. An and-box is *quiet* if it is solved and the constraint of the and-box does not restrict its environment (θ) outside the local variables of the and-box (or more formally $\mathcal{TC} \models \theta \supset \exists V((\wedge C) \wedge \theta)$, where $\mathcal{TC}$ is the current theory of constraints). Roughly, an and-box is quiet if it doesn't contain bindings for external variables for which there are no corresponding bindings in its environment.

3.3 Computation Model

The computation model is a transition system on configurations. Each kind of transition is defined by a conditional rewrite rule $G \Rightarrow G'$, that substitutes the goal G', on its right hand side, for some part of the configuration that matches the goal G, on its left hand side, if the associated condition is satisfied.

The *local forking rule*

$$A \Rightarrow \mathbf{choice}(\mathbf{and}(G_1)_{V_1} \ \%\ B_1, \ldots, \mathbf{and}(G_n)_{V_n} \ \%\ B_n)$$

rewrites a program atom A, not occurring in the body of a guarded goal, where A :- G_1 % $B_1, \ldots, A$:- G_n % B_n is the definition of the predicate of A, with the arguments of A substituted for the formal parameters, and the local variables of the i:th clause replaced by the variables in the set V_i. The sets V_i are chosen to be disjoint from the set of variables in the rewritten configuration.

The *determinate promotion rule*

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

may be applied if C_V is solved. If % is a pruning operator it is also required that C_V is quiet.

The *nondeterminate promotion rule*

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

can promote a wait-guarded goal with a solved guard, if R or S is non-empty and the rewrite is performed within a stable and-box. An and-box is *stable* if i) no other rule is applicable to any subgoal of the and-box, and ii) the and-box satisfies a constraint-stability condition.

A *constraint stability condition* is a condition from which it follows that no possible changes in the environment (through applications of rules to other subgoals of the global configuration) will lead to a situation in which non-trivial environment synchronisation, cut pruning, or commit pruning is applicable in the and-box. Note that a top-level and-box always satisfies a constraint stability condition, as it has no environment.

For the constraint system in this paper, a general constraint stability condition is that no environment of a subgoal of the and-box may restrict variables outside the and-box. In other words, this means that the and-box should not anywhere contain bindings of variables that are external to the and-box for which there are no corresponding bindings in its environment.

The *cut rule*

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

prunes to the right of a guarded goal with a quiet solved guard C_V.

The *commit rule*

$$\mathbf{choice}(R, C_V \mid B, S) \Rightarrow \mathbf{choice}(C_V \mid B)$$

prunes both to the right and the to left of a guarded goal with a quiet solved guard C_V.

The *environment synchronisation rule*

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

fails an and-box if the constraint of R is incompatible with the environment of the box.

The *guard distribution rule*

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

distributes the guard operation over a branch in an or-tree in the guard of a guarded goal.

The *choice elimination rule*

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

removes a failed guarded goal.

The *failure propagation rule*

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

fails an and-box, if it contains an empty choice-box.

Below we will define the aggregating operation *bagof*, which is applicable when the constraint system makes it possible to form lists. It is related to the bagof found in Prolog, but different in that it will not attempt to enumerate different solutions for different constraints on the external variables. Instead, the solutions are required to be quiet. This will allow us to retain their relation to external variables. The end result is a solution collecting operation reminiscent of the list comprehension operation of some functional languages.

Bagof-boxes are introduced by the following rule.

$$\text{bagof}(X, B, Y) \Rightarrow \mathbf{bagof}(X, \mathbf{and}(B)_{\{X\}}, Y)$$

We may rewrite bagof-boxes by

$$\mathbf{and}(R, \mathbf{bagof}(X, \mathbf{fail}, Y), S)_V \Rightarrow \mathbf{and}(R, Y = [], S)_V$$

$$\mathbf{and}(R, \mathbf{bagof}(X, \mathbf{or}(S_1, C_V, S_2), Y), T)_W \Rightarrow$$
$$\mathbf{and}(R, Y = [X' \mid Y'], C', \mathbf{bagof}(X, \mathbf{or}(S_1, S_2), Y'), T)_{V \cup W}$$

if C_V is quiet. The sequence of constraints C' is C in which the variables V that are local to the solution are given new names, occurring nowhere else in the configuration. In particular, the variable X is renamed to X'.

This completes the description of the computation model.

4 Basic Logic Programming Paradigms

In this section, we show that the basic logic programming paradigms of GHC and Prolog are available in AKL. As a notion such as programming paradigm is rather loosely defined, this is very much a matter of belief based on a set of convincing examples. We have convinced ourselves by adapting programs written in these languages and running them in our AKL implementation. Prolog programs often only require moving output unification after cut. GHC programs require no modification at all. PARLOG programs are almost as easy.

4.1 Subsuming GHC

A program written in the GHC subset of AKL will execute almost exactly as in GHC. In AKL, the commit operation is delayed until there exists a corresponding external binding for each local binding of an external variable. In GHC the binding operation itself is delayed until that moment. The difference in computational behaviour is that the binding will not be visible to the sibling goals in the guard in GHC, whereas it will be visible in AKL. The local bindings will sometimes detect failure, for example in

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

?- p(Z).
```

In AKL this would fail, whereas GHC would suspend. Worse is that GHC can arbitrarily 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 a context with the "otherwise" guard goal or a failure-detecting meta-call facility, as in KL1, the problem is of course more serious.

4.2 Subsuming "Prolog"

The first goal of this section is to show that the Prolog style of execution of pure definite clauses is easily achievable in AKL. Then it is shown how to translate Prolog definitions that contain cut. It is also shown how a simple improvement of the translation of pure clauses introduces the determinacy detecting, stream-and parallelisable style of execution given by the Andorra principle. Finally, it is shown how to make way for independent and-parallel execution.

Using data-flow analysis, it is possible to make translation from Prolog to AKL completely automatic along the lines outlined below [2].

4.2.1 Trivial Translation of Pure Definitions

The first translation considered completely ignores the possibility of putting goals in the guard. A pure definite clause with some terms t_i in the head

H(t_1, ..., t_n) :- B.

is translated as

H(X_1, ..., X_n) :- true : X_1=t_1, ..., X_n=t_n, B.

putting the equality constraints corresponding to head unification in the body. Note that the arguments of this AKL clause are distinct variables, and that the guard is empty.

Assume in the following the above trivial translation of the definitions of some predicates p, q, and r. The definition of p has two clauses

p :- true : B_1.
p :- true : B_2.

The execution of the goal

?- p, q, r.

proceeds as follows. The initial goal is an and-box containing the goal.

and(p, q, r)

Computation begins by local forking on each of the atomic goals, and with further computation within the resulting guarded goals. However, as the guards of the guarded goals for p are empty, these are immediately solved.

and(**choice**(**true** : B_1, **true** : B_2), **choice**(R), **choice**(S))

Assuming that the goals q and r have at least two candidate clauses, the and-box is also stable. If either goal had a single defining clause, this would be reduced first, and a similar stable state would be reached shortly.

Computation may now proceed with nondeterminate promotion of the body B_1, and then determinate promotion of the single remaining body B_2. The following state would then be achieved in which computation may proceed in the promoted bodies in both of the two branches.

$\mathbf{or}(\mathbf{and}(B_1, \mathbf{choice}(R), \mathbf{choice}(S)),$
$\quad \mathbf{and}(B_2, \mathbf{choice}(R), \mathbf{choice}(S)))$

It is fairly obvious that this style of translation leads to a computation which resembles SLD-resolution, and thereby the basic programming paradigm of pure Prolog.

Pure Horn-clause programs can also be executed by the following meta-interpreter, which is derived from the Or-parallel Prolog interpreter in Concurrent Prolog by Ken Kahn [11]. (The predicate clauses/2 returns a list of clauses for a goal, and append/3 is the directional append from two input lists to one output list.) The major difference is that the following is a full-fledged all-solutions interpreter as opposed to the single-solution interpreter in Concurrent Prolog. This is because we can use the wait-operator in resolve/3 instead of a commit-operator.

```
solve([]).
solve([A|As]) :- | clauses(A, Cs), resolve(A, Cs, As).

resolve(A, [(A :- Bs)|Cs], As) :-
        append(Bs, As, ABs), solve(ABs) : true.
resolve(A, [C|Cs], As) :-
        resolve(A, Cs, As) : true.
```

It is also possible to use this technique when translating a program.

4.2.2 Translating Definitions with Cut

The difference between cut in Prolog and cut in AKL is that cut is quiet in AKL. Therefore, output must be produced in the body-part of the clause, as in GHC.

```
Prolog: H(in,out) :- G, !, B.
AKL:    H(in,Y) :- G ! Y=out, B.
```

The quietness restriction is essential for parallel execution by making cut insensitive to the order of execution of goals. Also, with the quietness restriction, cut in AKL can be given an intuitive logical interpretation as "if-then-else"; see [5] for a discussion and proof. An important corollary is the soundness of negation-as-failure in its most general way (without being constructive) in AKL.

Unfortunately, the quietness restriction also makes some pragmatically justifiable programming tricks impossible 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.

As when translating from Prolog to a language such as GHC, there are work-arounds that do not involve noisy pruning. However, in some cases, the translation is quite non-trivial, and cannot be readily automated.

In the following, 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 example that will be used is the lookup-procedure as defined below. In Prolog it is simply written as follows.

```
lookup(X, [X|R]) :- !.
lookup(X, [Y|R]) :- lookup(X, R).
```

Its definition is equivalent to the well-known memberchk procedure, and when used for checking membership of a ground element in a ground list, its behaviour is equivalent in AKL. However, in Prolog it can also be used to add a new element to a list with an uninstantiated tail, and to find an element matching a given partially instantiated template.

```
| ?- lookup(key=value, Dict), lookup(key=Value, Dict).

Dict = [key=value|R],
Value = value ?
```

As mentioned above, this technique is not available in AKL. Now, there are several work-arounds for this problem. It is for example quite possible to manage a dictionary as a complete list of key-value pairs, as in the following.

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

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

When using this kind of dictionary, it is also necessary to pass the dictionary along in a more explicit way than for the Prolog solution. Note that access to the dictionary is serialised in the above solution. Two updating lookups can not go on at the same time in the same list. The following solution allows several lookups at the same time. It uses an incomplete list as in the Prolog solution, but access to the dictionary has to be managed by a dictionary server in order to synchronise the additions to the tail of the list.

```
dict(S) :-
        merger(S, S1), % (optional, see section 6.2)
        dictionaryserver(S1, D, D).

dictionaryserver([lookup(K, V)|S], D, T) :-
        | lookup(K, V, D, T, NT),
          dictionaryserver(S, D, NT).
dictionaryserver([], D, T).

lookup(K, V, [K=V1|R], T, NT) :- | V = V1, T = NT.
lookup(K, V, [K1=V1|R], T, NT) :- not(K = K1)
        | lookup(K, V, R, T, NT).
lookup(K, V, T, T, NT) :- | T = [K=V|NT].
```

This could have been an FGHC program, but 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.

Note that 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 some meta-logical primitive would be needed to achieve the same effect. The incomplete structure technique is of course more useful when the dictionary is organised as a tree. Unless sophisticated memory management techniques are used (such as reference counting, producer-consumer language restrictions (e. g. Janus),

or compile-time analysis), alternative solutions with complete structures will require O(log(N)) allocated memory for each new addition to the tree. The following Prolog program will only allocate the new node.

```
treelookup(X, t(X, L, R)) :- !.
treelookup(X, t(Y, L, R)) :- X @< Y, !, treelookup(X, L).
treelookup(X, t(Y, L, R)) :- treelookup(X, R).
```

In the AKL, the tree can be represented by a tree of processes, as in GHC. It can be argued that this is less efficient than the Prolog solution, but new results on compilation techniques for FGHC-like programs suggest that this inefficiency is not necessarily an inherent problem (see [12]).

A more definitive solution, allowing fully automatic translation, is to add noisy cut as a new guard operator in the language [8, 10]. This is an option in the Kernel Andorra Prolog framework, and it has been included in the prototype implementation of AKL. Noisy cut has the disadvantage of being less concurrent than quiet cut, and it also requires that programs are properly synchronised, to avoid problems with back-propagation of values. As it is mainly intended for backward compatibility in the context of automated translation from Prolog to AKL, it is omitted from this presentation.

Another difference from Prolog is that AKL, for simplicity, only allows definitions where the clauses are of the same kind. Mixed definitions must be translated into their corresponding unmixed definitions. Almost always, this means adding a cut to all clauses.

In some rare cases, a Prolog definition, e. g.

```
H :- B1.
H :- G, !, B2.
H :- B3.
```

has to be translated into two (or more) AKL definitions as follows

```
H :- true : B1.
H :- true : H1.

H1 :- G ! B2.
H1 :- true ! B3.
```

to make "backtracking" from B_1 to B_2 possible. Our experience of translating Prolog programs is that elegant translations are always available in the specific cases.

Finally, it should be noted that there is no fully general translation of cut inside a disjunction. However, it is the opinion of the authors that this abominable construct should be avoided anyway.

4.2.3 Andorra Principle Style Translation

The Andorra principle can be seen as a computation rule for pure definite clause programs. It states that atomic goals that have at most one candidate clause (determinate goals) should be selected first. Only when there are no such goals may another goal be selected (a nondeterminate goal) [14].

One appealing property is that all determinate goals may be executed in parallel, thereby extracting implicit dependent and-parallelism from pure programs. For example, the Andorra-I implementation provides both dependent and- and or-parallel execution on the Sequent Symmetry [3].

In AKL, this principle has been generalised to a language with deep guards, and is embodied in the stability condition. However, the basic principle itself is available as a special case. Its implications are further discussed in section 5.1. In this section, we present a translation from pure definite clauses to AKL. As a consequence of the AKL computation model, the translated programs will behave exactly as stated by the Andorra principle.

A clause can be considered to be a candidate clause for a goal, in the above sense, if its head unification and primitive test goals (such as $</2$ and the like) will not fail for the goal. By putting the head unification and the primitive test goals before a wait-operator, local execution in the guard will establish whether the clause fails for the goal or not. The and-box failure and the determinate promotion rule have priority over nondeterminate promotion, and therefore execution in AKL will be as given by the Andorra principle.

A pure definite clause

$p(t_1, \ldots, t_n)$:- Tests, Body.

can thus be translated to AKL as

$p(X_1, \ldots, X_n)$:- $X_1=t_1, \ldots, X_n=t_n$, Tests : Body.

according to the above suggested scheme. User-defined tests can of course also be counted among the candidate clause detecting guard goals.

4.2.4 Independent-AND Style Translation

When it is known that two goals are independent when executed by Prolog, a more interesting translation scheme is possible. The implications of this and related translations for programming search problems are discussed in section 5.2. Assume that in the Prolog 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 to AKL enables independent parallel execution of q and r as soon as X is sufficiently instantiated by its producers.

```
p(X) :- true : q1(X,Y), r1(X,Z), s(Y,Z).

q1(X,Y) :- q(X,Y1) : Y = Y1.
r1(X,Z) :- r(X,Z1) : Z = Z1.
```

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

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 [2].

4.3 Meta-Interpretation

Some logic programming languages allow compact meta-interpreters to be written. Meta-interpreters come to many different uses. A meta-interpreter can be written in AKL in the traditional style.

```
prove(true) :- | true.
prove((P,Q)) :- | prove(P), prove(Q).
prove(X=Y) :- | X=Y.
prove(A) :- guardoperator(A, wait) | trywait(A).
prove(A) :- guardoperator(A, cut) | trycut(A).
prove(A) :- guardoperator(A, commit) | trycommit(A).

trywait(A) :- clause((A :- G : B)), prove(G) : prove(B).

trycut(A) :- clause((A :- G ! B)), prove(G) ! prove(B).

trycommit(A) :- clause((A :- G | B)), prove(G) | prove(B).

guardoperator(append(A,B,C), wait).

clause((append(A,B,C) :- A = [], B = C : true)).
clause((append(A,Y,C) :- A = [E|X], C = [E|Z] : append(X,Y,Z))).
```

The try-procedures will allow nondeterminate generation within their guards quite independently, as no restrictions are imposed on external variables before promotion of the alternative clauses.

5 Controlling Nondeterminism

AKL not only subsumes Prolog and GHC, but it also provides new mechanisms for controlling search in nondeterministic programs.

5.1 Finite Domain Constraint Techniques

There are several examples showing the strength of the Andorra principle, and AKL in particular, for finite domain constraint programming. An often used example is the n-queens problem [13, 9, 1]. The following program is adapted from [13] using a finite domain constraints package written entirely in AKL.

```
queens(N, []).
queens(N, [X|Y]) :- fd(domain(N, X)), noattack(X, Y), queens(N, Y).

noattack(X ,Xs) :- noattack(X, Xs, 1).

noattack(X, [], Nb).
noattack(X, [Y|Ys], Nb) :-
        fd((X ≠ Y, X ≠ Y - Nb, X ≠ Y + Nb)), Nb1 is Nb+1,
        noattack(X, Ys, Nb1).
```

The domain(N, X) constraint generates values from 1 to N.

The corresponding program in CHIP has to be rewritten to provide generalised forward checking, but since AKL delays nondeterminate goals, the determinate goals will execute first, and all of the constraints will be available before alternatives are tried for the domain variables. Note that the domain is not predefined in this AKL solution, but is provided as a parameter. There is also no need to declare the domain variables. It is only necessary to call the domain generator.

5.2 Local Execution

The purpose of this example is to show how to execute two goals independently and completely, and how to collect their combined solutions in a Cartesian product manner. For simplicity, the more than familiar member-predicate is used.

It should be mentioned that virtually nothing is gained for this particular example when compared to its Prolog execution. However, when enumeration is costly, as in some search problems, the illustrated functionality is useful.

The following is the definition of member using the trivial AKL translation.

```
member(X, Y) :- true : Y=[X|Y1].
member(X, Y) :- true : Y=[Z|Y1], member(X, Y1).
```

The goal is to find the common members of two lists.

```
?- member(X, [1,2,3]), member(X, [2,3,4]).
```

We would like the two goals to be executed locally, and that the solutions are combined afterwards in a Cartesian product manner.

Complete local execution of a goal is achieved in AKL by putting the goal in the guard part of a clause, as shown in section 4.2.4. For this purpose we introduce an auxiliary predicate m, defined by

```
m(X,Y) :- member(X,Y) : true.
```

It encapsulates the call to member in a guard, in order to execute it locally. The previous goal is now restated.

```
?- m(X,[1,2,3]), m(X,[2,3,4]).
```

The execution proceeds as follows. The initial goal is an and-box containing the two atomic goals. All rewrites will be performed on both goals "in parallel".

$$\mathbf{and}(\mathrm{m}(X,[1,2,3]),\mathrm{m}(X,[2,3,4]))$$

First of all, local forking is applied to the m goals. The member goals now appear in the guard of a wait guarded goal.

$$\begin{array}{l}\mathbf{and}(\mathbf{choice}(\mathbf{and}(\mathrm{member}(X,[1,2,3])) : \mathrm{true}),\\ \quad\mathbf{choice}(\mathbf{and}(\mathrm{member}(X,[2,3,4])) : \mathrm{true}))\end{array}$$

The first step of the local execution is local forking of the member goals with the two clauses in the definition, producing choice-boxes with two guarded goals each with trivially solved guards.

$$\begin{aligned}&\textbf{choice}(\textbf{and}(\textbf{choice}(\textbf{true} : X = 1,\\ &\qquad\qquad\qquad \textbf{true} : \text{member}(X, [2,3]))) : \text{true},\\ &\qquad \textbf{and}(\textbf{choice}(\textbf{true} : X = 2,\\ &\qquad\qquad\qquad \textbf{true} : \text{member}(X, [3,4]))) : \text{true})\end{aligned}$$

As this state is stable, nondeterminate promotion may be applied on either of the branches, followed by determinate promotion of the other. (There is room for some indeterministic choice between rules here, but with equivalent result. The most lucid execution is shown.)

$$\begin{aligned}&\textbf{and}(\textbf{choice}(\textbf{and}(X = 1) : \text{true},\\ &\qquad\qquad \textbf{and}(\text{member}(X, [2,3])) : \text{true}),\\ &\qquad \textbf{choice}(\textbf{and}(X = 2) : \text{true},\\ &\qquad\qquad \textbf{and}(\text{member}(X, [3,4])) : \text{true}))\end{aligned}$$

Now note that the member goals that have been promoted are similar in appearance to the previous member goals. It is easily seen that the last two steps can be repeated two more times for each of the remaining elements in the lists. Finally, there will appear a member(X,[]) goal that will fail and disappear. The final result after completed local execution is as follows.

$$\begin{aligned}&\textbf{and}(\textbf{choice}(\textbf{and}(X = 1) : \text{true}, \textbf{and}(X = 2) : \text{true}, \textbf{and}(X = 3) : \text{true}),\\ &\qquad \textbf{choice}(\textbf{and}(X = 2) : \text{true}, \textbf{and}(X = 3) : \text{true}, \textbf{and}(X = 4) : \text{true}))\end{aligned}$$

This state is stable. We may now apply nondeterminate promotion on the first choice-box, promoting all its local solutions.

$$\begin{aligned}&\textbf{or}(\textbf{and}(X = 1, \textbf{choice}(\textbf{and}(X = 2) : \text{true}, \textbf{and}(X = 3) : \text{true},\\ &\qquad\qquad\qquad\qquad \textbf{and}(X = 4) : \text{true}),\\ &\qquad \textbf{and}(X = 2, \textbf{choice}(\textbf{and}(X = 2) : \text{true}, \textbf{and}(X = 3) : \text{true},\\ &\qquad\qquad\qquad\qquad \textbf{and}(X = 4) : \text{true}),\\ &\qquad \textbf{and}(X = 3, \textbf{choice}(\textbf{and}(X = 2) : \text{true}, \textbf{and}(X = 3) : \text{true},\\ &\qquad\qquad\qquad\qquad \textbf{and}(X = 4) : \text{true}))\end{aligned}$$

The "grid" of the Cartesian product should now be apparent. Applying environment synchronisation and failure propagation rules yields the final answers.

$$\textbf{or}(\textbf{and}(X = 2), \textbf{and}(X = 3))$$

Hopefully, this example provided some intuition for the style of local execution available in AKL.

5.3 Structured Nondeterminism

Between doing no local computation, as in SLD-resolution, and completely local computation, as in the previous section, lies a whole spectrum of possible organisations of nondeterministic computations. The tradeoff is on one side between the danger of repeating work, as when only little work is done locally in advance, and the danger of wasting work, as when complete failure destroys the local work that has been done. Also, the more that is done locally, the more may be available for parallel execution on a multi-processor.

The following example shows how a useful tradeoff can be achieved (adapted from [15]). It is a program that finds common sublists of two lists. In its first formulation, there is only trivial local execution.

```
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]).
```

If the trivial or the Andorra principle style translation is used, local execution is very limited. It is reasonably safe to execute each goal locally until the first point at which the execution for this goal could fail in an interesting way. This happens when the first element of a sublist is generated.

To achieve this effect, the above definition is transformed into the following.

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

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

The following illustrates the state that will be reached when executing the above goal. The generation is basically performed on the members of the lists. Thus, we start with the initial goal

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

After a while the following configuration is reached.

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

Subsequent nondeterminate promotion will combine the solutions for the local or-trees, producing the following tree of independent alternatives.

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

This formulation is able to share some part of the computation of the second goal between the solutions for the first goal.

6 Controlling Reactive Computations

The ability to encapsulate nondeterminate computation within an otherwise reactive computation opens up for new programming techniques.

6.1 Encapsulated Nondeterminism

The AKL model makes it possible to encapsulate a process doing nondeterminate search within a process performing a determinate computation in a way that allows both processes to proceed concurrently in a fair implementation of the language. Determinacy can be enforced by the encapsulating goal either by the use of a pruning operator, or by the use of bagof.

6.2 Multi-way Merge with Constant Delay

The following realises a multi-way merger with constant delay that uses encapsulated nondeterminism. It is called with one initial input stream and one initial output stream. The input stream can be split by binding it to a term split(S1,S2). The merger consists of two cooperating goals: 1) a generator that detects that one of the streams being merged has become instantiated and then enters it on a stream to a server, 2) a server that for each instantiated stream either removes a message (and feeds the rest to the generator), splits the stream (and feeds the two new streams to the generator), or closes the stream.

```
merger(In, Out) :-
        server(B, A, Out, s(0)),
        bagof(S, generator(S, [In|A]), B).
```

The generator procedure enters the streams as nondeterministic alternative solutions for the bagof goal. It requires that the streams are bound, and bagof will wait until they are.

```
generator(S, [S1|R]) :- | generator(S, S1, R).

generator(S, S, R) :- stream(S).
generator(S, S1, R) :- generator(S, R).

stream([]) :- |. stream([E|R]) :- |. stream(split(S1,S2)) :- |.
```

The server procedure expects a stream of instantiated streams from the bagof goal. Apart from dealing with these in the above mentioned way, it also keeps track of the number of merged streams, and closes the output stream and terminates the bagof goal when none remain.

```
server([[E|R]|B], A, S, N) :-
        | S = [E|S1], A = [R|A1], server(B, A1, S1, N).
server([split(S1,S2)|B], A, S, N) :-
        | A = [S1,S2|A1], server(B, A1, S, s(N)).
server([[]|B], A, S, s(N)) :-
        | server(B, A, S, N).
server(B, A, S, 0) :-
        | A = [], S = [].
```

The binding of A to [] in the last clause will make the generator fail, whereby the bagof is terminated.

7 Discussion

AKL is being implemented and the results so far are promising. The current implementation covers everything described in this paper. In our AKL environment, we have been able to use public domain code originally written in Prolog (by O'Keefe and others) with minor modifications only. Nondeterministic benchmarks have also been easy to adapt. Most committed-choice examples run with no modifications except simple changes to the syntax.

When compared to a high-performance Prolog implementation with an abstract machine that is emulated in C (e. g. SICStus), the current implementation executes deterministic code about four times slower. Experiments show that a flat guard optimisation alone will bring this factor down to between 1.5 and 2. There is nothing in the abstract machine itself that is substantially less efficient than the WAM when executing deterministic code. Therefore, it is expected that performance will be close to that of a WAM implementation of Prolog.

Nondeterminate promotion is currently implemented using copying, as in the rule itself. Naive copying seems to incur an overhead of between 30% and 60% on the execution of highly nondeterministic problems such as N-queens. We are experimenting with more clever copying schemes that avoid copying of boxes that would fail almost immediately. Experiments show that this can be very beneficial.

Acknowledgements

We would like to thank Torkel Franzén and Johan Montelius of the Andorra group at SICS, and also the other 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 the Sixth International Conference on Logic Programming.* MIT Press, 1989.

[2] Francisco Bueno and Manuel Hermenegildo. Towards a translation algorithm from Prolog to the Andorra Kernel Language. PEPMA Internal Report, January 1991.

[3] Vitor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the Basic Andorra model. Technical note, University of Bristol, Department of Computer Science, March 1990.

[4] Torken Franzén. Logical aspects of the Andorra Kernel Language. (Revised version of SICS Research Report R90008, submitted to ILPS'91).

[5] Torken Franzén. Formal aspects of the Andorra Kernel Language: I. Research Report R90008, SICS, May 1990.

[6] Seif Haridi. A logic programming language based on the Andorra model. *New Generation Computing*, (7):109–125, 1990.

[7] Seif Haridi and Per Brand. Andorra Prolog, an integration of Prolog and committed choice languages. In *Proceedings of the FGCS*, 1988.

[8] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its computation model. In *Proceedings of the Seventh International Conference on Logic Programming*. MIT Press, 1990. (Revised version of SICS Research Report R90002).

[9] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1990.

[10] Vijay A. Saraswat and Seif Haridi. Some notes on Andorra Prolog. unpublished note, May 1989.

[11] Ehud Shapiro. Concurrent Prolog: A progress report. *IEEE Computer*, 8(19):44–58, August 1986.

[12] Kazunori Ueda. A new implementation technique for flat GHC. In *Proceedings of the Seventh International Conference on Logic Programming*. MIT Press, 1990.

[13] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[14] David H. D. Warren. The Andorra principle. Presented at the Gigalips workshop, Stockholm, 1987.

[15] David H. D. Warren. The Extended Andorra Model with implicit control. presented at a Parallel Logic Programming workshop in Eilat, June 1990.

[16] Rong Yang and Hideo Aiso. P-Prolog: A parallel logic langauge based on exclusive relation. In *Proceedings of the Third International Conference on Logic Programming*. MIT Press, 1986.

A workshop record for the Parallel Logic Programming workshop in Eilat, with copies of slides and position papers, is available from SICS.