

- [11] Sverker Janson and Johan Montelius. Design of a sequential prototype implementation of the Andorra Kernel Language. SICS research report, Swedish Institute of Computer Science, 1992.
- [12] W. Kornfeld. Constraint programming in Andorra Prolog. Presented at the Swedish-Japanese-Italian workshop, 1989.
- [13] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, Tokyo, Japan, November 1988.
- [14] Michael J. Maher. Logic semantics for a class of committed choice programs. In *Proc. of the Fourth International Conference on Logic Programming*. MIT Press, 1987.
- [15] V.A. Saraswat. The concurrent logic programming language CP: definition and operational semantics. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–63. ACM, New York, 1987.
- [16] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989. To be published by the MIT Press.
- [17] V.A. Saraswat. Programming in Andorra Prolog. Technical report, Xerox PARC, 1989.
- [18] Kazunori Ueda. Guarded Horn clauses. Technical Report TR-103, ICOT, June 1985.
- [19] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
- [20] D. H. D. Warren. The Andorra principle. Presented at the Gigalips workshop, Stockholm, 1987.
- [21] R. Yang. Solving simple substitution ciphers in Andorra-I. In Giorgio Levi and Maurizio Martelli, editors, *Proc. of the Sixth International Conference on Logic Programming*, Lisboa, 1989. MIT Press.

Acknowledgements

The visit at SICS of Catuscia Palamidessi, during which this work was carried out, was supported by the Andorra project at SICS.

The authors wish to thank Vijay Saraswat, Torkel Franzén, D.H.D. Warren and Bill Kornfeld for many valuable comments and suggestions. This work is part of ESPRIT Project 2471 (“PEPMA”), and is supported by the Swedish National Board for Industrial and Technical Development (NUTEK), Swedish Telecom, and Ericsson Group.

References

- [1] Khayri A. M. Ali and Roland Karlsson. The Muse approach to or-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.
- [2] R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In Giorgio Levi and Maurizio Martelli, editors, *Proc. of the Sixth International Conference on Logic Programming*, pages 471–486. MIT Press, 1989.
- [3] Vitor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the Basic Andorra model. In *Proc. of the Eighth International Conference on Logic Programming*. 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] Torkel Franzén, Seif Haridi, and Sverker Janson. An overview of the Andorra Kernel Language. In *Proc. of the 2nd Workshop on Extensions to Logic Programming*. Springer-Verlag, 1992.
- [6] Steve Gregory. *Parallel Logic Programming in PARLOG*. Addison-Wesley, 1987.
- [7] S. Haridi and P. Brand. Andorra Prolog:, an integration of Prolog and committed choice languages. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 745–754, Tokyo, 1988. Institute for New Generation Computer Technology (ICOT).
- [8] Seif Haridi, Sverker Janson, Johan Montelius, and Martin Nilsson. Ports for objects in concurrent logic programs. SICS research report, Swedish Institute of Computer Science, 1992.
- [9] Seif Haridi and Catuscia Palamidessi. Structural operational semantics of Kernel Andorra Prolog. In *Proc. of the Conference on Parallel Architectures and Languages Europe (PARLE)*. Springer Verlag, 1991.
- [10] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *Proc. of the 1991 International Logic Programming Symposium*. MIT Press, 1991.

```

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

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.

```

or(and( $L = []$ )_{L},
    or(fail,
        or(and( $L = [a|L1]$ , sublist( $L1, [t, s]$ ), sublist( $L1, [s, t]$ ))_{L,L1},
            or(and( $L = [t|L1]$ , sublist( $L1, [s]$ ), sublist( $L1, []$ ))_{L,L1},
                and( $L = [s|L1]$ , sublist( $L1, []$ ), sublist( $L1, [t]$ ))_{L,L1}))))
```

We now have one solved and-box, with the solution $L = []$, and one failed and-box, which had assumed that L would start with c . Execution will now continue within each of the unsolved and-boxes, replacing them with or-trees analogous to the one above, but we will stop here, hopefully having illustrated the idea of an AKL computation.

13 Related Work

Vijay Saraswat has over a number of years been studying languages similar to AKL, a study culminating in the family of concurrent constraint languages [15, 16], and a lot of inspiration for AKL comes from his work, including the view of constraints, and his programming techniques for concurrent languages with don't know nondeterminism.

A major difference is that our control specifies *when* to promote don't know nondeterminism, which makes programming substantially easier. Also, we emphasize fully interleaved execution in a language with deep guards. However, AKL is definitely a concurrent constraint language in its spirit.

14 Discussion

We presented a formal transformational semantics for AKL. The semantics is transformational since it describes the final results of a computation (as a set of abstract trees), and there is no notion of interaction with the environment.

The semantics given treats and-boxes seen at the outermost level as black boxes until the box either succeeds, fails or suspends. This notion is not sufficient. In particular, reactive AKL programs, which do not exhibit top-level don't know nondeterminism, need a more refined notion where actions in an and-box can affect its environment, i.e. some sort of a reactive semantics, and a notion of observational equivalence based on it.

Other more important issues—like efficient implementation, on both single- and multi-processor architectures, and programming methodology—have the highest priority, and our effort is currently devoted to them [11].

12 An AKL Example

In this example, also used in [10] and [5], we illustrate both the incremental growth of or-trees, as exploited in the operational semantics in the previous section, and how the notion of local execution in guards makes it possible to write elegant search-programs. It is a program that finds common sublists of two lists. We start with a Prolog program for the same task.

```
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 goal. 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.

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

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

Let us start with the initial goal

```
and(sublist(L, [c, a, t, s]), sublist(L, [l, a, s, t]))_{L}
```

The sublist goals are first unfolded into choice-boxes.

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

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.

```
and(choice(and(L = []) : true,
            and(L = [c|L1])_{L1} : sublist(L1, [a, t, s]),
            and(L = [a|L1])_{L1} : sublist(L1, [t, s]),
            and(L = [t|L1])_{L1} : sublist(L1, [s]),
            and(L = [s|L1])_{L1} : sublist(L1, [])),
```

Let (P, \leq) be an poset (partially ordered set). A *directed set* in P is a subset D of P such that

$$\forall a, b \in D \exists c \in D [a \leq c \wedge b \leq c].$$

An *ideal* S is a directed set which is *downward closed*, i.e. such that

$$\forall a \in S [b \leq a \Rightarrow b \in S].$$

The set of ideals of P , ordered by set inclusion, we will denote by $(Id(P), \subseteq)$. It is well known that it is a Complete Partial Order (i.e. it has a minimum, and each non-empty set of elements admits a least upper bound) and it contains a sub-CPO isomorphic to (P, \leq) . $(Id(P), \subseteq)$ is called *completion by ideals* of (P, \leq) . The elements of the subset isomorphic to (P, \leq) will be denoted by the corresponding elements of P .

Definition 11.3 (The Domain of Interpretation) *Our domain of interpretation is the complete partial order $(\mathcal{T}^\omega, \leq)$ (the domain of finite and infinite or trees), which is the completion by ideals of the poset (\mathcal{T}, \leq) . The least upper bound of a directed subset $\mathcal{D} \in \mathcal{T}^\omega$ will be denoted by $\sqcup \mathcal{D}$.*

Definition 11.4 (Operational Semantics) *The operational semantics of a program is a function $\mathcal{O}: \text{IGoal} \rightarrow \mathcal{P}(\mathcal{T}^\omega)$, where IGoal is the set of all initial goals and $\mathcal{P}(\mathcal{T}^\omega)$ is the set of all the subsets of \mathcal{T}^ω . \mathcal{O} is defined as follows.*

$$\mathcal{O}(P) = \{\sqcup_i \mathcal{O}'(P_i) : i \in \{0, 1, 2, \dots\}, P \equiv P_0, P_i \xrightarrow[\text{true } \emptyset]{A} P_{i+1}\}$$

Note that the environment of the whole configuration is always empty. $\mathcal{O}': \text{Goal} \rightarrow \mathcal{T}^\omega$ is defined as follows:

- $\mathcal{O}'(\mathbf{deadlock}) = \mathbf{deadlock}$.
- $\mathcal{O}'(\mathbf{fail}) = \mathbf{fail}$.
- $\mathcal{O}'(\mathbf{or}(G, G')) = \mathbf{or}(\mathcal{O}'(G), \mathcal{O}'(G'))$.
- $\mathcal{O}'(\mathbf{and}(R)_V) = \begin{cases} \exists V. \sigma(R) & \text{if } R = C, \text{ and } C \text{ is satisfiable} \\ \perp & \text{otherwise} \end{cases}$

Note that, if

$$P_0 \xrightarrow[\text{true } \emptyset]{A} \dots P_i \xrightarrow[\text{true } \emptyset]{A} \dots$$

then $\{\mathcal{O}'(P_i)\}_{i \geq 0}$ is a chain. Therefore (since a chain is a particular case of directed set), the definition of \mathcal{O} is correct.

The rules for suspension cover all the cases in which the computation rules are not applicable, excepting the or-boxes. Namely, a configuration is final only if it is of one of the following forms: **fail**, **deadlock**, an and-box of the form C_V , or an or-box containing only final configurations. This means that the semantics of a configuration is always a set maximal objects (possibly infinite) in \mathcal{T}^ω (i.e., the leaves are not labeled by \perp).

11 Operational Semantics

We now present the definition of the operational semantics of AKL in set-theoretic terms. Intuitively, the operational semantics of a program is defined by the semantics of the *initial goals*, of the form $\mathbf{and}(R)_{\text{var}(R)}$, that can be run under that program, and the semantics of an initial goal consists of the set of all the possible collections of answers that can be obtained by running it.

In general, all the answers that are delivered under certain particular nondeterminate choices are collected in a goal expression with nested or-boxes at the top-level. The or-boxes form a tree-structure, with the intermediate nodes labeled by **or** and the leaves are labeled and-boxes. The and-boxes not of the form C_V correspond to computations that have not yet terminated, and we can define the semantics as the limit of the or-trees obtained along a certain computation (transition) chain. This limit can, in general, be an infinite tree, as the computation can deliver an infinite set of answers. All the information about the result of a computation is preserved in the semantic structure, and it leaves space for further abstractions.

Indeed, different implementations may lead to different notions of *observables*. For instance, we may imagine a situation similar to Prolog, in which the answers are presented in the order they are collected by the usual depth-first strategy. A loop along one branch will cause the unobservability of the answers possibly generated at the right of that branch. To model this, we can abstract from our semantics the sequence of answers obtained by collecting left-to-right the leaves of the tree corresponding to a successfully terminated computation. The sequence ends when we get in correspondence of a nonterminating and-box.

Another possibility is to collect in parallel all the answers generated, without any restriction on the order in which they appear. To model this we can abstract from our semantics the set of the leaves corresponding to successfully terminated computations.

Definition 11.1 *The set \mathcal{T} (the set of or trees) is the minimal set that satisfies the following conditions:*

- $\perp \in \mathcal{T}$
- **fail, deadlock** $\in \mathcal{T}$
- if θ is satisfiable then $\exists V.\theta \in \mathcal{T}$
- if $t, t' \in \mathcal{T}$ then **or**(t, t') $\in \mathcal{T}$

The set \mathcal{T} is ordered as follows

Definition 11.2 *The relation \leq is the minimal ordering relation on \mathcal{T} that satisfies the following conditions*

- $\forall t \in \mathcal{T}. \perp \leq t$
- $\forall t, t', u, u' \in \mathcal{T}. (t \leq u \wedge t' \leq u') \supset \mathbf{or}(t, t') \leq \mathbf{or}(u, u')$

- A nondeterminate transition is admissible if it obeys the restrictions defined in the previous section.

$$\frac{G \xrightarrow[\theta U]{N} G'}{G \xrightarrow[\theta U]{A} G'} \quad \text{if } \text{ndcond}(G, \theta, U)$$

- An admissible transition of a stable box can be propagated to outer goals.

$$\frac{G \xrightarrow[\theta \wedge \tau U \cup W]{A} G'}{C[G] \xrightarrow[\theta U]{A} C[G']} \quad \text{if } \tau_W = \text{env}(C[\])$$

This completes the description of the AKL transition system. We now proceed to characterize the notion of deadlock.

10 Suspension Rules

For the semantic characterization, we also want to identify *deadlocked* goals. The following rules describe deadlock, and the propagation of deadlock from the inner goals to the outer ones. Admissible computations are extended accordingly.

Deadlock

A goal not of the form C_V deadlocks whenever it is stable and there is no available nondeterminate transition.

$$G \xrightarrow[\theta U]{A} \mathbf{deadlock} \quad \text{if } \text{stable}(G) \text{ and } \neg \exists G' (G \xrightarrow[\theta U]{N} G')$$

Choice-Boxes

If the the guard operator is $|$ or $:$, then a choice-box deadlocks whenever all the branches become deadlocked.

$$\frac{G_1 \xrightarrow[\theta U]{A} \mathbf{deadlock}, \dots, G_n \xrightarrow[\theta U]{A} \mathbf{deadlock}}{\mathbf{choice}(G_1 \% A_1, \dots, G_n \% A_n) \xrightarrow[\theta U]{A} \mathbf{deadlock}} \quad \text{if } \% \in \{ |, : \}$$

If the the guard operator is $!$, then a choice-box deadlocks whenever the first branch becomes deadlocked.

$$\frac{G \xrightarrow[\theta U]{D} \mathbf{deadlock}}{\mathbf{choice}(G ! A, \dots) \xrightarrow[\theta U]{D} \mathbf{deadlock}}$$

And-Boxes

An and-box deadlocks whenever all the internal local goals deadlock.

$$\frac{G_1 \xrightarrow[\tau U]{D} \mathbf{deadlock}, \dots, G_n \xrightarrow[\tau U]{D} \mathbf{deadlock}}{\mathbf{and}(G_1, \dots, G_n)_V \xrightarrow[\theta W]{D} \mathbf{deadlock}} \quad \text{if } \begin{array}{l} \tau = \theta \wedge \sigma(G_1, \dots, G_n) \\ U = W \cup V \end{array}$$

9.1 Nondeterminacy and Stability

The BAM strategy of performing determinate goals in preference to nondeterminate goals translates to a necessary condition for nondeterminate transitions, viz. that no determinate transitions are applicable to the and-box.

AKL, with its “deep” computations, needs a more elaborate condition. Nondeterminate transitions are to be admissible only if determinate transitions cannot become possible as a result of extending the environment of constraints and variables. Thus, the condition does not depend on transitions in the (implicit) context of the goal, and in more intuitive terms we can say that the condition is not time-dependent.

A goal G is *stable* relative to an environment θ and V iff

- i) no determinate (D) transition is applicable to G in this environment, and
- ii) no satisfiable extension of the environment can lead to a goal in which a determinate (D) transition is applicable to G .

More formally, the property $stable(G, \theta, V)$ is defined as follows.

$$stable(G, \theta, V) \equiv \neg \exists G' \exists U \exists \tau (V \cup U) [U \cap \text{var}(G) = \emptyset \wedge G \xrightarrow[\theta \wedge \tau]{D} V \cup U G']$$

Where we by $\tau(W)$ mean a constraint only containing variables in W .

We will now proceed to state the precise conditions on nondeterminate promotion. In addition to stability, we require that the transition should be on the *innermost* stable candidate for nondeterminate promotion in the following sense.

$$\begin{aligned} \text{ndcond}(G, \theta, V) \equiv & \\ & stable(G, \theta, V) \wedge \\ & \neg \exists C \exists X [G = C[X] \wedge X \neq G \wedge \text{env}(C[\]) = \tau_U \wedge stable(X, \theta \wedge \tau, V \cup U) \wedge \\ & \quad \exists X' (X \xrightarrow[\theta \wedge \tau]{N} V \cup U X')] \end{aligned}$$

Additional conditions may be imposed on nondeterminate promotion, stipulating e.g. that nondeterminate promotion must be applied with respect to the leftmost possible solved guard, or that it must be applied to an innermost eligible candidate within the stable box. Such further restrictions will not be considered in this presentation.

In an implementation of AKL one will need to replace the abstract definition of stability by a more manageable sufficient condition for stability. For the standard constraint system of finite trees (often called Herbrand) such a condition is simple both to define and implement [11].

9.2 Admissible Computations

The AKL computation model is defined by the following transition system (based on the previous one). The transition relations are labelled by A and describe *admissible* transitions.

- A determinate transition is always admissible

$$\frac{G \xrightarrow[\theta U]{D} G'}{G \xrightarrow[\theta U]{A} G'}$$

- $\text{var}(\sigma) = \emptyset$
- $\text{var}(p(x_1, \dots, x_n)) = \emptyset$
- $\text{var}(\mathbf{or}(G_1, G_2)) = \text{var}(G_1) \cup \text{var}(G_2)$
- $\text{var}(\mathbf{and}(G_1, \dots, G_n)_V) = V \cup \text{var}(G_1) \cup \dots \cup \text{var}(G_n)$
- $\text{var}(\mathbf{choice}(G_1, \dots, G_n)) = \text{var}(G_1) \cup \dots \cup \text{var}(G_n)$
- $\text{var}(G \% A) = \text{var}(G)$

The rule can now be stated as

$$\frac{G \xrightarrow[\tau W]{m} G'}{\mathbf{and}(R, G, S)_V \xrightarrow[\theta U]{m} \mathbf{and}(R, G', S)_V} \quad \text{if } \begin{array}{l} \tau = \theta \wedge \sigma(R, S) \\ W = U \cup \text{var}(\mathbf{and}(R, S)_V) \end{array}$$

9 Control of the Computation Model

A sequence of applications of the transition rules described in the previous section constitutes an unrestricted derivation or *computation* of the extended model. AKL computations are restricted in a manner to be described in this section. The control of AKL was motivated (and described informally) in [10].

To formalize this restriction, we will need two more concepts: that of a *context*, which is a goal with a “hole”, and that of an *environment of a context*, which is the environment of constraints and variables of the hole.

The symbol $[]$ denotes the hole, and the expression $C[]$ denotes a context. We define contexts inductively as follows.

- $[]$ is a context.
- if $C[]$ is a context then $\mathbf{or}(G, C[])$, $\mathbf{or}(C[], G)$, $\mathbf{and}(R, C[], S)_V$, and $\mathbf{choice}(R, (C[] \% A), S)$ are contexts.

The expression $C[G]$ denotes the goal obtained by substituting a goal G for $[]$ in $C[]$.

The next definition formalizes the notion of *environment of a context*. We define a function $\text{env}(C[]) = \theta_V$, which returns the environment of constraints θ and variables V of the hole of $C[]$.

- $\text{env}([]) = \mathbf{true}_\emptyset$
- $\text{env}(\mathbf{or}(G, C[])) = \text{env}(\mathbf{or}(C[], G)) = \text{env}(C[])$
- $\text{env}(\mathbf{and}(R, C[], S)_V) = (\theta \wedge \sigma(R, S))_{W \cup V}$, where $\theta_W = \text{env}(C[])$
- $\text{env}(\mathbf{choice}(R, (C[] \% A), S)) = \text{env}(C[])$.

Commit Promotion

$$\mathbf{choice}(R, C_V \mid A, S) \xrightarrow[\theta U]{D} \mathbf{choice}(C_V \mid A)$$

if R or S is non-empty and C_V is satisfiable and quiet (as above).

Nondeterminate Promotion

$$\mathbf{and}(T_1, \mathbf{choice}(R, C_V : A, S), T_2)_W \xrightarrow[\theta U]{N} \\ \mathbf{or}(\mathbf{and}(T_1, C, A, T_2)_{V \cup W}, \mathbf{and}(T_1, \mathbf{choice}(R, S), T_2)_W)$$

if R or S is non-empty.

Guard Distribution

$$\mathbf{choice}(R, \mathbf{or}(G, G') \% A, T) \xrightarrow[\theta U]{D} \mathbf{choice}(R, G \% A, G' \% A, T)$$

8 Structural Rules

The following rules allow us to derive the transitions of goals depending on the transitions that can be made by the components of the goals.

Or-Boxes

Any transition made by a goal inside an or-box is propagated to the parent box.

$$\frac{G \xrightarrow[\theta U]{m} G'}{\mathbf{or}(R, G) \xrightarrow[\theta U]{m} \mathbf{or}(R, G')} \\ \mathbf{or}(G, R) \xrightarrow[\theta U]{m} \mathbf{or}(G', R)$$

Choice-Boxes

Any transition made by a goal inside a choice-box is propagated to the parent box.

$$\frac{G \xrightarrow[\theta U]{m} G'}{\mathbf{choice}(R, G \% A, S) \xrightarrow[\theta U]{m} \mathbf{choice}(R, G' \% A, S)}$$

And-Boxes

Any transition made by a goal inside an and-box, models the fact that the environment of the goal consists of the environment and the constraint of the parent and-box.

This rule is restrained by the condition that the new local variables introduced in a transition of a subgoal must be disjoint from the variables of the other subgoals. This will ensure that all the local variables of different components are always disjoint, thus avoiding clashes of variables. We therefore introduce the notion of *variables local to a goal* $\text{var}(G)$.

2. σ is *quiet w.r.t. θ and V* iff $\mathbf{TC} \models \theta \supset \exists V(\sigma \wedge \theta)$.

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

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 .

7 Computation Rules

Local Forking

If A is a program atom then

$$A \xrightarrow[\theta U]{D} \mathbf{choice}(\mathbf{and}(G_1)_{V_1} \% A_1, \dots, \mathbf{and}(G_n)_{V_n} \% A_n)$$

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 a “renaming” set V_i , for which $V_i \cap U = \emptyset$.

The structural rules of the transition system will guarantee that the new variables V_i are different from the variables of the global configuration.

Environment Synchronization

$$\mathbf{and}(R)_V \xrightarrow[\theta U]{D} \mathbf{fail}$$

if $\sigma(R) \wedge \theta$ is unsatisfiable.

Failure Propagation

$$\mathbf{and}(R, \mathbf{fail}, S)_V \xrightarrow[\theta U]{D} \mathbf{fail}$$

Choice Elimination

$$\mathbf{choice}(R, \mathbf{fail} \% A, S) \xrightarrow[\theta U]{D} \mathbf{choice}(R, S)$$

Determinate Promotion

$$\mathbf{and}(R, \mathbf{choice}(C_V \% A), S)_W \xrightarrow[\theta U]{D} \mathbf{and}(R, C, A, S)_{V \cup W}$$

if $\sigma(C)$ is satisfiable, and in case $\%$ is cut or commit is also quiet w.r.t. $\theta \wedge \sigma(R, S)$ and V .

Cut Promotion

$$\mathbf{choice}(R, C_V ! A, S) \xrightarrow[\theta U]{D} \mathbf{choice}(R, C_V ! A)$$

if S is non-empty and C_V is satisfiable and quiet (as above).

grammar.

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

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

We let $Goal$ denote the set of expressions generated by $\langle goal \rangle$. In the following, the letters R , S , and T stand for sequences of goals, A for a sequence of atoms (occasionally a single atom), 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 of goals will be written using comma, which is not likely to be confused with the use of comma for separating arguments. The letters V , W , and U stand for finite sets of variables.

The symbol **fail** will be used to denote the empty or-box and the empty choice-box, regarded as collapsing to the same object.

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. This information is necessary to deal with execution and suspension of the constraint operations. An and-box of the form $\mathbf{and}(C)_V$ will normally be written as C_V .

In a guarded goal, the goal 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 7 is applied to the goal.

The AKL transition system to be defined is a pair $\langle Conf, \longrightarrow \rangle$, where $Conf = Goal \cup \{\mathbf{deadlock}\}$, and \longrightarrow is a class of transition relations on $Conf$. The statement $G \xrightarrow[\theta U]{m} G'$ means that it is possible to make a transition from G to G' in the mode m . The subscripts θ and U are called the *environment of constraints and variables* of the transition. The letters D , N , and A , stand for *determinate*, *nondeterminate*, and *admissible* mode, respectively. Constraints are discussed in the next section.

6 Constraints

Constraints will be regarded as formulas in some constraint language. The only constraint formulas that occur in goals are atomic constraints. 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.

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

1. σ is *satisfiable* iff $\mathbf{TC} \models \exists\sigma$,

general concurrent language—AKL. The computation model is defined in terms of a transition system.

First, we will present a set of computation rules and structural rules that define the various (determinate and nondeterminate) computation steps that can be performed. Then we discuss the issue of how the control of BAM generalises to the new setting, giving us the notion of *stability*. This allows us to define the precise control of AKL as the *admissible* computation steps.

Having defined the computation model, we then define the set-theoretic operational semantics. This semantics will characterize computations in terms of their “results”. A computation can either succeed, deadlock, or be non-terminating, and the nondeterminate steps will produce an or-tree of computations, all of which may have different results. We choose to identify deadlocked computations in terms of additional computation rules, called suspension rules.

In the following sections we will define the syntax of clauses and goals (Section 5), present our view of constraints (Section 6), define the computation rules (Section 7), define the structural rules (Section 8), make precise the control (Section 9), define the suspension rules (Section 10), and define the operational semantics (Section 11). Finally, we show an example AKL program (Section 12).

5 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. More about constraints in Section 6. It is assumed that these two classes of atomic formulas are disjoint. The remaining syntactic categories pertaining to programs are the following.

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

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.

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

The execution of an AKL program will be represented as a series of transitions between goal expressions. The syntax of these expressions is defined by the following

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.

1	0	0	0
0	0	V ₂₃	V ₂₄
0	V ₃₂	0	V ₃₄
0	V ₄₂	V ₄₃	0

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 (V₁₁) is given the value 1. It appears in the row V₁₁ to V₁₄, in the column V₁₁ to V₄₁, and in the diagonal V₁₁ to V₄₄. 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 V₁₂ and V₂₄ are assumed to contain queens, and propagation of the above kind has taken place.

0	1	0	0
0	0	0	1
V ₃₁	0	0	0
V ₄₁	0	V ₄₃	0

Here we examine the propagation that this state will give rise to. Notice that in row 3, all variables but V₃₁ 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 V₄₁) the value 0. Finally, V₄₃ is given the value 1 by reasoning as above.

The above program is 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 [19]. 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 tried to give the flavor of the style of problem solving that is made available by BAM, and therefore by AKL. The (simple) mapping of definite clause programs to AKL to achieve the effect of BAM is outside the scope of this paper; these details can be found in [10].

4 The Extended Computation Model

We will now proceed to define the extended computation model, which takes advantage of the principles underlying the BAM to control nondeterminacy in a more

xcell/3 goals and ycell/3 goals are determinate if the first argument is known, or if the second two arguments are known to be 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 squares 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

The BAM model divides a computation into *determinate* and *nondeterminate phases*. First, all atomic goals for which it is known that at most one clause would succeed are reduced using a single clause during the determinate phase. (These goals can be reduced in arbitrary order, and also in and-parallel.) Then, when no such goal is left, some goal is chosen, typically the leftmost, for which all clauses are tried; this is called the nondeterminate phase. The computation then proceeds with a determinate phase on each or-branch.

An atomic goal is said to be *determinate* when there is at most one candidate clause that can succeed for the goal. Whenever it becomes known that an atomic goal is determinate, the goal can either be reduced, if a single clause would apply, or it can fail, if it is known that no clause would apply. It is not considered to be an error if the mechanism for detecting the determinacy of goals fails to detect that a goal is determinate. In general, nothing less than complete execution will establish this property.

The BAM model has a number of interesting properties.

First, the BAM allows determinate goals to be run in and-parallel, extracting implicit and-parallelism from the program.

Second, the notion of determinacy in the BAM gives a reasonably strong form of *synchronization*. As long as a goal is able to produce data deterministically, no consumer of this data is allowed to run ahead (if it does not know what to consume). This allows specification of concurrent processes.

Third, the BAM reduces the search space by executing the determinate goals first. Goals can fail early, and the constraints produced by a reduction can reduce the number of alternatives for other goals. This has been proved to be very relevant for the coding of constraint satisfaction problems [12, 17, 2, 7, 21], and this we will try to show in the next section.

3 An Andorra Example

Although it is not the main purpose of this paper to present programming techniques in AKL, we still feel that we should argue for our choice of language by showing an elegant AKL program, also used in [5], which exploits the BAM for efficient constraint satisfaction. We have attempted to make this section self-contained for readers with some prior acquaintance with logic programming.

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 a new algorithm. The novelty lies in the way the algorithm is expressed. Let us therefore first familiarize ourselves with some particulars of AKL, especially how the notion of determinacy is employed.

Consider the following two predicates, which will be the building blocks in our N-queens program.

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

The control principles of BAM make possible a simple combination of don't know nondeterministic search and communicating processes, simply by always performing deterministic work first. In addition to the potential for or-parallel execution of search as in Prolog [13, 1], BAM provides a potential for and-parallel execution, similar to that of concurrent logic languages such as GHC and Parlog. In many Prolog programs, both of these forms of parallelism can be exploited to a high degree [3].

The concurrent constraint framework is a foundation for various kinds of work on concurrent languages. We have adopted the basic notions related to constraints, such as satisfiability and entailment, somewhat generalising their use in AKL.

By dealing explicitly with concurrency, AKL becomes a true concurrent programming language, rather than a parallelizable sequential language, while still providing the powerful problem solving capability given by don't know nondeterminism. Through the refined control scheme, AKL is a considerable improvement over Prolog in this regard. This capability will be exemplified in Section 3 and in Section 12. These examples were chosen because they illustrate those aspects of the language that are peculiar to AKL. In addition, AKL provides the programming paradigms of Prolog, the committed-choice language, and the concurrent constraint languages, of which of which there are many illustrations in the literature.

The basic notions of AKL are conjunction (corresponding to parallel composition) and guarded clauses (corresponding to guarded statements). There are three forms of guarded clauses, corresponding to three forms of selection. Clauses guarded by the *cut* operator correspond to if-then-else conditionals. Clauses guarded by the *commit* operator correspond to the original guarded statements, with their don't care nondeterministic choice. Clauses guarded by the *wait* operator provide don't know nondeterministic selection, where each alternative can be tried in separate "worlds".

Admittedly, the understanding of this exposition would benefit from prior acquaintance with AKL as given in [10]. However we have tried to make the paper reasonably self contained, within the space allotted. In particular, we have tried to make the exposition "juicier" by showing examples of AKL programs, although they do not as such contribute to the presentation of AKL semantics.

This paper is a thoroughly revised version of [9], which was based on the Kernel Andorra Prolog framework, from which AKL is derived.

2 The Basic Andorra Model

As a preliminary, we define the Basic Andorra Model (BAM) for pure definite clauses. It demonstrates an essential feature of the AKL control, namely that of giving priority to deterministic computation over nondeterministic computation.

We will henceforth use the terms *determinate* and *nondeterminate*, instead of deterministic and nondeterministic. A computation step which is determinate does not involve guessing, e.g. of alternative clauses in a reduction step, although there might still be a nondeterministic choice between several alternative determinate steps. A computation step which is nondeterminate involves guessing (with the intention to explore all alternatives if necessary), although the choice of which step to perform might be quite deterministic.

Structural Operational Semantics for AKL*

Seif Haridi Sverker Janson
Swedish Institute of Computer Science[†]

Catuscia Palamidessi
Centre for Mathematics and Computer Science[‡] and
Department of Computer Science, Utrecht University[§]

Abstract

The Andorra Kernel Language (AKL) is a concurrent constraint programming language. It can be seen as a general combination of logic programming languages such as Prolog, GHC, and Parlog, the first of which provides don't know nondeterminism, and the last two of which are concurrent logic programming languages. The constraint system is an independent parameter of the language description.

In this paper, we revisit the description of Janson and Haridi [10], adding the formal machinery which is necessary in order to completely formalize the control of the computation model. To this we add a formal description of the transformational semantics of AKL. The semantics is a set of *or-trees* which also captures infinite computations.

1 Introduction

The Andorra Kernel Language (AKL) is a general combination of search-oriented don't know nondeterministic logic programming languages, such as Prolog, and process-oriented concurrent logic programming languages, such as GHC, Parlog, and others [18, 6, 14, 16]. For an introduction to the language from this perspective, see [10]. For a treatment from a logical point of view, see [4].

Obviously, AKL is biased towards symbolic processing, by heritage, by providing don't know nondeterminism, and by the central role of constraints, but our goal for AKL is that it should also be useful as a general purpose concurrent language for shared memory multi-processor architectures. These aspects are emphasized elsewhere, where we show that AKL is potentially a concurrent object-oriented language par excellence [8].

AKL has one half of its roots in the Basic Andorra Model (BAM) for definite clause programs, proposed by Warren [20, 7], and the other half in the concurrent constraint framework developed by Saraswat [16].

*Also in Journal of Future Generation Computer Systems, special issue on selected papers from PARLE'91.

[†]Box 1263, S-164 28 KISTA, Sweden, email: {seif,sverker}@sics.se

[‡]P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, email: katuscia@cw.nl

[§]P.O. Box 80089 3508 TB Utrecht, The Netherlands