Design for AKL with Intelligent Pruning

Salvador Abreu and Luís Moniz Pereira

Departamento de Informática
Universidade Nova de Lisboa
2825 Monte de Caparica
PORTUGAL
{spa,lmp}@fct.unl.pt

Abstract. In this article we discuss issues posed by the introduction of dependency-based search space pruning in an And-Or Tree Rewriting System (AORS), such as AKL and the Extended Andorra Model. A mechanism relying on two simple devices (the alt and other attributes) is presented. The construction and application of alt and other is illustrated in the context of the configuration rewrite rules for AKL.

1 Introduction and Motivation

The Andorra Kernel Language's (AKL, see [10]) computational model addresses the matter of handling nondeterminism in a different way than Prolog, ie. a computation does not explore the solution space by exploring a derivation tree but rather by deploying a sequence of configurations (see [9] for an introduction to this subject). AKL configurations have one significant difference wrt. Prolog derivation trees which is, that the ancestor hierarchy is flattened.

AKL (and the Andorra family of and-or parallel Prolog systems) already makes good use of two search-space narrowing techniques:

- The Andorra Principle, related to the "Sidetracking" search procedure of [14], which consists in focussing attention first on goals known to be determinate.
- Constraint propagation.

It may therefore be questionable whether it is worthwhile increasing the overhead of the runtime execution of these systems in order to obtain a shorter proof (in terms of number of inference steps). We argue that it is, because both search-space narrowing techniques built into the AKL model rely exclusively on forward execution, not on any knowledge of whatever occurred during backward execution, especially the information conveyed by the occurrence of failures. Our previous work in this direction (see [1, 2]) shows that there is a non-negligible gain to be obtained from the failure information, even with "simple-minded" heuristics.

Our present work explores techniques for pruning the search-space so as to obtain *consistent* effective speedups for combinatorial problems, while retaining an execution efficiency close to that of the original system. To achieve this we propose to make use of binding dependency information to guide the unfolding of an AKL computation. One way this is accomplished is by improving the accuracy of the methods explored in [2].

The basic problem of intelligent backtracking (IB) in Prolog (see [4, 6]) is to locate the bindings responsible for the occurrence of a failure, and backtrack directly to their originator, designated as the culprit, so that unnecessary work (ie. irrelevant to the conflict) is eschewed. To do so, schemes to implement IB in Prolog (for example see [4, 12, 5, 7]) always refer to the Prolog runtime data structures used to implement nondeterminism, namely choice-points; ie. a binding is said to depend on a given goal if it is associated with that goal's choice-point.

AKL provides no such concept, nondeterminism being handled differently, by the use of *choice-boxes* which are one of the constructors for configurations. Also, it should be noted that the context of both variables and bindings in AKL is the other configuration constructor, the *and-box*. Our approach will tie bindings susceptible of being revised to the entities which may provide alternative bindings: sets of choice-boxes; it also provides a few other structuring concepts lacking in the computational model. We call our system AKL/IP for "AKL with Intelligent Pruning".

2 AKL with Intelligent Pruning

Our model develops along two major lines, embodied in two devices absent in the original AKL computational model:

- **Alternative generators.**
  These associate to each binding susceptible of being revised a reference to its alternative generator. The latter designates a set of configuration constructors, and may point to either a choice-box or an and-continuation.[1]
  The binding of a variable V to a value X in AKL has an associated *binding environment* designated as $\mathsf{env}(V)$, which is simply an and-box. In AKL/IP variable bindings will have an additional component, the *alternative generator*, designated as $\mathsf{alt}(V)$ which is a reference to a choice-box or an and-continuation.
  The alt attribute is also defined for terms in general, by the introduction of artificial "hidden" variables, wherever a term may occur: for instance the binding of variable X to a compound term $X = p(a, b)$ can be seen as the set of bindings $\{X = p(V_1, V_2), V_1 = a, V_2 = b\}$. Corresponding to each equality in this set there will be an alt which specifies (through transitive closure) the parts of the configuration that are susceptible of providing an alternative binding for the variable in question. This makes it possible to trace any given term's history.
  AKL/IP choice-boxes and and-continuations also have an alt component, and it is because of this that the alt attribute of bindings can be seen as designating a *set* of potential alternative generators.

[1] An and-continuation is the residual form of an alternative in a choice-box.

- **Shared binding information.**
  Shared bindings are the result of the choice split operation, and even though the current AKL implementation (AKL/PS 0.8) does not provide actual sharing, the information is necessary in order to determine where a given binding is being used, outside the context of its environment (or home and-box, $\mathsf{home}(V)$).
  The purpose of this is to allow us to forcibly remove from the configuration all and-boxes that involve a binding that is known to be invalid. This occurs as the result of a conflict's originator being a specific binding; it is then important to be able to locate all instances of that particular binding in order to fail the and-boxes where it occurs.
  For this purpose we define a new component for bindings, and call it other. $\mathsf{other}(V)$, where V is a bound variable, will be a set of and-boxes. All the members of $\mathsf{other}(V)$ share the binding of V with $V's$ environment $\mathsf{home}(V)$. This comes about as a result of a choice-split operation.

With these concepts present, we may now proceed to describe the modified configuration transformations under AKL/IP. This is done in the next two sections.

3 Forward Execution – Data structure construction

The AKL configuration transformations dealing with forward execution are *local forking*, *choice split* and *determinate promotion*. All of these affect the alt and other components of bindings and choice-boxes; we now detail these:

- **Local Fork**.
  Given a subgoal C, and considering that the predicate designated by C has the clauses $H_1 \leftarrow G_1\%B_1, \ldots, H_n \leftarrow G_n\%B_n$, where % is a guard operator, this operation replaces the call by a choice-box containing the alternative clauses, where clause i is represented as $\mathsf{and}_{\mathcal{V}_i}(G_i)\,\%B_i$, meaning an and-box containing the guard G_i, with an attached continuation which stands for the body of the clause, B_i. The new variables introduced by this clause are denoted by $\mathcal{V}_i$.

  $$\begin{array}{l} \quad \mathsf{and}_{\mathcal{V}}(\cdots C \cdots) \\ \Rightarrow \mathsf{and}_{\mathcal{V}}(\cdots \mathsf{choice}(\mathsf{and}_{\mathcal{V}_1}(G_1)\,\%B_1, \cdots \mathsf{and}_{\mathcal{V}_n}(G_n)\,\%B_n) \cdots) \end{array}$$

  When a binding to a variable in $\mathcal{V}$ (or above) occurs from within one of the inner and-boxes, for example inside G_i in the partial configuration $A_i = \mathsf{and}_{\mathcal{V}_i}(G_i)\,\%B_i$, execution in A_i will proceed to the end of the guard (G_i), and then not be allowed to continue past the guard operator because doing so would require the application of a nondeterminate step.

Additionally, if % is a pruning guard operator (i.e. cut or commit), execution will *suspend* because the guard is required to be quiet in these cases.[2]

- **Choice Split**.

  The choice split is the operation that accounts for don't-know nondeterminism in AKL. In a choice split, one alternative is singled out from a set of choices.

$$
\begin{array}{l}
\quad\mathsf{choice}\left(\begin{array}{c}\cdots\\ \mathsf{and}_{\mathcal{V}}(\cdots\mathsf{choice}(\cdots A_{i-1},\mathsf{and}_{\mathcal{V}_i}(G_i)\,\%B_i,A_{i+1}\cdots)\cdots)\\ \cdots\end{array}\right)\\
\Rightarrow\mathsf{choice}\left(\begin{array}{l}\qquad\qquad\cdots\\ A_1=\mathsf{and}_{\mathcal{V}}(\cdots C_1=\mathsf{choice}_{C_2}(\mathsf{and}_{\mathcal{V}_i}(G_i)\,\%B_i)\cdots)\\ A_2=\mathsf{and}_{\mathcal{V}'}(\cdots C_2=\mathsf{choice}(\cdots A_{i-1},A_{i+1}\cdots)\cdots)\\ \qquad\qquad\cdots\end{array}\right)
\end{array}
$$

  Since this is the step where a choice is made among alternatives, the selected one must refer to the other ones as its alternative generator. This is acheived by setting $\mathsf{alt}(C_1)\leftarrow C_2$. Variables in $\mathcal{V}$ will have their alt default to C_2. Initially $\mathcal{V}'=\mathcal{V}$, denoting the environment copy operation, where all variables local to the and-box being duplicated also get cloned. At this point the bindings in $\mathcal{V}'$ must be marked as copies of those in $\mathcal{V}$ so that we can later locate the various places where a given binding is being used, ie. for all bindings $V\in\mathcal{V}$ we'll set $\mathsf{other}(V)\leftarrow A_2$.

- **Determinate Promotion**.

  The determinate promotion step selects a choice-box with a single and-box below it (the *promoted* and-box), and merges it with the parent and-box (the *promoted-to* and-box). Such a configuration may be the direct consequence of a choice split transformation.

  In a determinate promotion variables variables local to the promoted and-box are imported back into the promoted-to and-box, and bindings to variables external to the promoted and-box are installed in the promoted-to and-box. Variables local to the promoted-to and-box that were bound as external in the promoted and-box are revised to become local bindings.

$$
\begin{array}{l}
\quad\mathsf{and}_{\mathcal{V}}(\cdots C_{i-1},C_i=\mathsf{choice}(\mathsf{and}_{\mathcal{V}_s}()\,\%B)\,,C_{i+1}\cdots)\\
\Rightarrow\mathsf{and}_{\mathcal{V}\cup\mathcal{V}_s}(\cdots C_{i-1},B,C_{i+1}\cdots)
\end{array}
$$

  The promoted and-box already has executed to the end of the guard, and therefore consists of a set of local variables and bindings thereto, constraints on external variables, and an and-continuation. All bindings being promoted (i.e. those in $\mathcal{V}_s$) will have $\mathsf{alt}(C_i)$ added to their alt. This is necessary because up to that point, all variables $V\in\mathcal{V}_s$ had an implicit alternative generator: the one for the immediately containing choice-box C_i, and this must now be

[2] A quiet guard is one that binds no external variables; a noisy guard is one that is not quiet.

made explicit.
Another approach would be to let $\mathsf{alt}(B) \leftarrow \mathsf{alt}(C_i)$ and add B (an and-continuation) to the promoted bindings' alt.

– **Cut and Commit.**
The *cut* and *commit* steps are the pruning guard operations. These remove alternatives from the containing choice-box.
The cut rule:

$$\begin{array}{l} \mathsf{choice}(\cdots A_{i-1}!B_{i-1}, A_i = \mathsf{and}_{\mathcal{V}_i}()!B_i, A_{i+1}!B_{i+1}, \cdots) \\ \Rightarrow \mathsf{choice}(\cdots A_{i-1}!B_{i-1}, A_i = \mathsf{and}_{\mathcal{V}_i}()!B_i) \end{array}$$

The commit rule:

$$\begin{array}{l} \mathsf{choice}(\cdots A_{i-1}|B_{i-1}, A_i = \mathsf{and}_{\mathcal{V}_i}()\,|B_i, A_{i+1}|B_{i+1}, \cdots) \\ \Rightarrow \mathsf{choice}(A_i = \mathsf{and}_{\mathcal{V}_i}()\,|B_i) \end{array}$$

The cut rule prunes to the right of the solved and-box, while the commit rule prunes on both sides. The commit rule is AKL's mechanism to implement don't-care nondeterminism.
These rules may only apply if the solved guard is quiet, i.e. a noisy guard will not be allowed to proceed: it will suspend until the external variable it tried to bind is instantiated, at which time it may resume execution.
The effect these operations have on the alt and other relations is that of removing all references to the parts of the configuration being pruned.

4 Backward Execution – Use of the data structures

It is during backward execution, ie. the occurrence of failure, that the information constructed during forward execution can be put to use, for instance by reordering the configuration.

The AKL configuration transformations that deal with backward execution are *environment synchronization*, *choice elimination* and *failure propagation*. These also affect the alt components.

– **Environment Synchronization.**
This is the basic failure operation. It amounts to:

$$A = \mathsf{and}_{\mathcal{V}}(G) \Rightarrow \mathsf{fail}$$

Whenever G is a constraint operation that fails because of an existing binding in $\mathcal{V}$ (eg. G could be a unification), the and-box containing the failing operation will be removed from the configuration (ie. replaced with fail).
Suppose the operation that failed was the unification of the two terms X_1 and X_2. Let $A_1 = \mathsf{alt}(X_1)$ and $A_2 = \mathsf{alt}(X_2)$; we apply the following algorithm:

1. If both A_1 and A_2 are null, meaning that this was a determinate goal, we simply follow the normal AKL behaviour, ie. the and-box will collapse. Exit from the pruning algorithm.

2. If only one of A_1 and A_2 is null, the conflict involves one revisable binding, call it A_R. Proceed to point 4.
3. If neither A_1 nor A_2 is null, the conflict arose from two nondeterminate bindings (ie. bindings with potential alternative values). Pick one of A_1 and A_2 and call it A_R. Continue to point 4.

   Note: the selection of A_R should pick the one that produced the youngest value of $\{X_1, X_2\}$, so as to explore all the clauses for the predicate that is failing in the same order as the unmodified AKL engine. This issue is not as crucial as is the case with Intelligent Backtracking in Prolog, because the AKL computational model will preserve completeness as it does not impose a strict bottom-up left-to-right order on the exploration of a derivation tree: choosing to revise the older binding does not exclude re-using its current value, just that the order would be altered.
4. Let the binding corresponding to A_R be denoted by X_R. We now collapse all and-boxes sharing X_R, thereby eliminating the conflict. In other words, let A be the environment of the failing goal G (i.e. the current inner and-box), for all $A' \in \{A\} \cup \mathsf{other}(X_R)$, remove A' from the configuration. Proceed to point 5.
5. Reorder the configuration.

   At this stage, we may want to reorder the AKL configuration as per [2]. The most significant difference here is that we can now bring the re-incident instances of the failing goal to immediately after the producer of the conflicting binding, which is arguably the optimal location, as is discussed in the aforementioned article.

   This should only be done when just one of the bindings is non-deterministic, ie. with a non-empty alt. The re-incident instances of the producer of that binding should not be relocated, while the re-incident instances of the determinate binding (the one with the empty alt) should be moved to after the former.

– **Choice elimination.**

  This operation removes failure nodes (fail and-boxes) from choice-boxes. It does not affect and is not affected by either alternative generators or shared bindings, so we retain the normal AKL behaviour.

$$\begin{array}{l} \quad \mathsf{choice}(\cdots A_{i-1}, \mathsf{fail}, A_{i+1} \cdots) \\ \Rightarrow \mathsf{choice}(\cdots A_{i-1}, A_{i+1} \cdots) \end{array}$$

– **Failure propagation.**

  Similarly to the choice elimination transformation, this operation does not involve bindings in any way. We perform it in the normal AKL manner.

$$\mathsf{and}(\cdots \mathsf{fail} \cdots) \Rightarrow \mathsf{fail}$$

5 Implementation Issues

As a first approach, and for simplicity's sake, the representation of bindings and configuration constructors (and-boxes, choice-boxes and and-continuations) will be extended to include linked lists for the alternative generators and the shared bindings. Support for the re-incident instances of goals is already available, from the prototype described in [2].

It is in our plans to optimize these data structures in order to make them implicit as far as possible, to avoid the overhead of removal from the linked lists on failure, as well as the extra computational effort required during garbage collection, since these structures are independent from the configuration tree, and therefore require some form of pointer relocation.

6 Conclusion

We are presently testing and evaluating the implementation and so cannot yet provide any experimental results. However, the results from [2] seem to indicate that the re-incident instance relocation alone should provide significant speedups for highly combinatorial programs. It is our expectation that having a dependency-directed scheme will eliminate the apparently random effect that the methods described in [2] can have on some programs, thereby contributing to the general usefulness of a programming system based on AKL/IP.

The techniques developed herein can be made to apply to a more general class of execution models for Logic Programming, that we designate generically as "And-Or Tree Rewriting Systems." These include AKL, but also the computational models derived from Kernel Andorra Prolog [9] and the Extended Andorra Model [17].

Acknowledgements

We would like to thank Philippe Codognet for various fruitful discussions and Torkel Franzén for his constructive review of an earlier version of this article. JNICT Portugal is also acknowledged for their financial support.

References

1. Salvador Abreu, Luís Moniz Pereira, and Philippe Codognet. Improving Backward Execution in Non-deterministic Concurrent Logic Languages. In Ryuzo Hasegawa and Mark Stickel, editors, *Workshop on Automated Deduction, Fifth Generation Computer Systems*. Institute for New Generation Computing, 1992.
2. Salvador Abreu, Luís Moniz Pereira, and Philippe Codognet. Improving backward execution in the andorra family of languages. In Apt [3], pages 384–398.
3. Krzysztof Apt, editor. *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Washington, USA, 1992. The MIT Press.

4. M. Bruynooghe and L. M. Pereira. Deduction revision by intelligent backtracking. In J.A. Campbell, editor, *Implementations of Prolog*. Ellis-Horwood, 1984.
5. C. Codognet and P. Codognet. Non-deterministic stream AND-Parallelism based on intelligent backtracking. In Levi and Martelli [13], pages 63–79.
6. Christian Codognet, Philippe Codognet, and Gilberto File. Yet another intelligent backtracking method. In Kowalski and Bowen [11], pages 447–465.
7. Philippe Codognet and Thierry Sola. Extending the WAM for intelligent backtracking. In Furukawa [8], pages 127–141.
8. Koichi Furukawa, editor. *Proceedings of the Eighth International Conference on Logic Programming*, Paris, France, 1991. The MIT Press.
9. Seif Haridi and Sverker Janson. Kernel andorra Prolog and its computation model. In Warren and Szeredi [18], pages 31–46.
10. Sverker Janson and Seif Haridi. Programming paradigms of the Andorra kernel language. In Saraswat and Ueda [15], pages 167–186.
11. Robert A. Kowalski and Kenneth A. Bowen, editors. *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seatle, 1988. ALP, IEEE, The MIT Press.
12. V. Kumar and Y.-J. Lin. An intelligent backtracking scheme for Prolog. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 406–414, San Francisco, August - September 1987. IEEE, Computer Society Press.
13. Giorgio Levi and Maurizio Martelli, editors. *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon, 1989. The MIT Press.
14. L. M. Pereira and A. Porto. Intelligent backtracking and sidetracking in horn clause programs. Technical Report CIUNL 2/79, Universitade Nova de Lisboa, 1979.
15. Vijay Saraswat and Kazunori Ueda, editors. *Logic Programming, Proceedings of the 1991 International Symposium*, San Diego, USA, 1991. The MIT Press.
16. Ehud Shapiro, editor. *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, London, 1986. Springer-Verlag.
17. David H. D. Warren. The extended andorra model with implicit control. ICLP90 Preconference Workshop, June 1990.
18. David H. D. Warren and Peter Szeredi, editors. *Proceedings of the Seventh International Conference on Logic Programming*, Jerusalem, 1990. The MIT Press.

This article was processed using the LaTeX macro package with LLNCS style