Improving Backward Execution in Non-Deterministic Concurrent Logic Languages

Salvador Abreu[*]
Luís Moniz Pereira
Departamento de Informática
Universidade Nova de Lisboa
2825 Monte de Caparica
PORTUGAL
spa,lmp@fct.unl.pt

Philippe Codognet
INRIA-Rocquencourt
BP 105, 78153 Le Chesnay
FRANCE
Philippe.Codognet@inria.fr

Abstract

We present a new mechanism for improving the execution of non-deterministic concurrent logic languages, such as the Andorra family of languages. The basic idea is, upon failure and backtracking, to re-schedule continuation goals in order to first execute the goal that has failed. In this way, if the backtrack point is not pertinent to the failure, the original failing goal will fail again and this will immediately amount to further deeper backtracking. Such a heuristic will hence save useless deduction/backtracking work. This method would be very difficult to implement in sequential stack-based logic systems such as Prolog, but fits well in the execution models of concurrent logic languages. We have implemented this scheme by modifying the Andorra Kernel Language prototype implementation developed at SICS (AKL version 0.0), and evaluation results show that this backward execution strategy improves performance for a variety of benchmarks, giving speedups up to a factor two or three.

1 Introduction

Since the first research in the early 80's, the field of concurrent logic programming has raised a number of important issues about the way of executing logic programs and how to move away from the traditional, simple but rigid, search strategy popularized by Prolog. On the one hand, the depth-first left-to-right strategy of Prolog, due to its simple stack-based execution model, has lead to tremendous advances in Prolog implementation technology which has been decisive for establishing logic programming as a mature declarative paradigm. Such results were exported to the area of automated deduction with theorem provers such as Stickel's PTTP. But on the other hand, one may sometimes prefer a more flexible control strategy, and a more adequate search strategy that reduces the overall number of inferences needed to solve the problem can make up for a slower inference speed. This is especially true for theorem proving and automated deduction applications, were the control may be as important as the inference speed in order to achieve good performances. In that direction concurrent logic languages such as KL1, Concurrent Prolog or Parlog are a step forward to escape from the rigid depth first search strategy. Moreover, some implementations of concurrent languages now achieve raw speeds close to that of traditional Prolog system.

[*]This work was done while at INRIA on an extended visit.

The *Andorra model* was proposed by D.H.D. Warren [15] in order to combine Or- and And-parallelism, and it has now bred a variety of idioms and extensions developed by different research groups, among which Andorra-I [3, 2] and Andorra Kernel Language [5, 6]. The essential idea is to execute determinate goals first and concurrently, delaying thus the execution of non-determinate goals until no determinate goal can proceed any more. Hence the Andorra principle can amount to *a priori* pruning and it can thus reduce the size of the computation when compared to standard Prolog up to one order of magnitude [2].

Although much work has been done to improve the forward execution in deductive systems, as exemplified by the Andorra model described above, very few works deal with the improvement of backward execution. It is however interesting, upon failure during the computation process, to take into account the information of this very failure to improve the search strategy. For instance, one can consider that the goal that has currently failed is likely to fail again after backtracking, that is, if the backtrack point is not pertinent. This goal should thus be executed first after backtracking, as this could immediately amount to further deeper backtracking if the backtrack point is not pertinent. In this way, the whole recomputation of the part of the proof between the backtracking point and the failing goal is avoided w.r.t a standard (non-reordering) strategy. The point being that if this goal just failed, it is likely to be "problematic" and therefore it may fail again with the next bindings produced as a result of backtracking. In other words, *a failed goal is likely to fail again.* this can indeed be seen as yet another instance of the "first-fail" principle apply to backward execution.

Such a re-ordering of the continuation goals (goals that remain in the resolvent) upon backtracking, based only on the information of which is the current failing goal, can thus save much deduction/backtracking work. This simple idea has indeed been first proposed by Lee Naish for some years[12] as an example of heterogeneous SLD resolution. A related computation rule is presented, and it is argued that it is a form of intelligent backtracking. However, such a strategy is not well suited to the computational model of Prolog as implemented for example by stack-based architectures such as the famous Warren Abstract Machine (WAM) because a failing goal is immediately discarded, leaving no trace, and it is very hard if not impossible to relate it to its other occurrences, which will be tried anew after the last choice-point's goal has generated another solution. There is also no way to manipulate and keep part of the stacks to avoid recomputing independent parts of the proof. Also remark that Prolog's operational semantics requires a strict execution order which prevent such re-ordering. One had to wait until the development of concurrent logic languages to fully rework those ideas. Indeed at the semantic level these languages have the necessary flexibility and do not impose a strict ordering, and at the implementation level they include sufficiently versatile data-structures to make re-ordering (re-scheduling) easy, as, in one way or an other, the whole computation tree has to be explicitly represented.

We have implemented this scheme in the prototype version of the Andorra Kernel Language (AKL 0.0) developed at SICS [7]. As mentioned above, one of the main interest of the proposed scheme is its ease of implementation by simple modifications of an existing implementation of a concurrent language. One has only to integrate a re-scheduling of continuation goals upon backtracking. There is no need to modify terms or variable bindings representations as in an intelligent backtracking scheme. The overhead of our machinery w.r.t. the original AKL implementation is very small, limited to 5-10%. We have experimented with various strategies on a variety of benchmarks, and we have encouraging results: a average gain of 50% (as measured by the number of nondeterminate promotions) can be observed, with peak values up to 300%.

2 Andorra Kernel Language

The semantics of the Andorra Kernel Language (AKL) is given by a set of rewrite rules applicable to and-or trees designated as *configurations*, see [5]. We will follow in this paper the AKL terminology for and-or trees: a subtree whose root is an or-node will be called a choice-box, and a subtree whose root is an and-node will be called a and-box.

The current (sequential) implementation of AKL [7] uses copying for the *choice splitting* operation, as decribed by the semantic rewrite rule. This operation is very different from the run-time organization of, say, Prolog systems where choice splitting, i.e. non-determinism is handled by choice-point creation and backtracking. In the AKL scheme, there are no choice-points, the concept being replaced by the multiple clauses of the nondeterminate goal being present in otherwise similar and-box contexts, contained in a common choice-box. This leads us to define the notion of copied or re-incident node as follows.
A *re-incident node* G' is another instance of a choice-box or and-continuation for a goal G that occurs in another branch, at the same depth in the and-or tree. In other words, given configuration (1),

$$\mathbf{choice}(\cdots, \mathbf{and}(S_1, \mathbf{choice}(\cdots), S_2), \cdots) \tag{1}$$

after applying the choice-splitting rule we obtain configuration (2).

$$\mathbf{choice}(\cdots, \mathbf{and}(S_1, \mathbf{choice}(\cdots), S_2), \mathbf{and}(S_1', \mathbf{choice}(\cdots), S_2'), \cdots) \tag{2}$$

S_1' and S_2' are *re-incident nodes* of respectively S_1 and S_2, as they share the same potential solutions, represented by an identical set of applicable clauses.

3 And-Or tree transformations based on failure

Our backward execution method will thus be formalized as an and-or tree transformation executed upon failure. Intuitively, the and-or tree transformations can be viewed as a heuristic to speed up the search space scanning process by providing a means by which separate or-branches of the proof tree influence one another: one branch's early failure may be used to "draw attention" to the part of the tree that caused failure, so that it will be selected earlier than it normally would in other instances present in other parts of the tree.

We experimented with different strategies for reordering the and-or tree, that can be presented under several orthogonal aspects, as described below.

1. **Under what circumstances to reorder?**

   These are conditions that may have to be fulfilled in order for the modifications to the shape of the and-or tree to take place. Different conditions are enumerated and may have to be satisfied independently of each other.

   (a) *When the failing node is an instance of the last clause for the parent goal.*
   This corresponds to reorder only in case of deep backtracking.

   (b) *When the re-incident nodes have the same grandparent choice-box*[1] *as the failing node.*
   The effect of this is to further narrow the scope of the reordering operation.

2. **What parts of the tree to reorder?**

   (a) *Reorder only the re-incident nodes.*

   (b) *Reorder all the nodes in a path from each re-incident node to the lowest common ancestor.*
   Using this criterion will make more extensive changes to the and-or tree, as it calls attention not only to a specific node's other instances, but also to the computation that gave rise to its failure.

[1] The grandparent choice-box is the parent choice-box of the containing and-box, ie. the closest enclosing choice-box.

3. **Where to reorder to?**

   If a a failure occurs and a given subset of the conditions specified in 1 are fulfilled, the current and-or tree (the AKL configuration) should be rewritten. The transformation will only affect the re-incident nodes of the failing literal, which will be moved. The possibilities we have considered are:

   (a) One interesting location to move a re-incident node to, assuming it failed because of the binding of a variable X, is right after X's producer. This is the optimal place to move it to, however it would require some form of variable binding dependency analysis, which puts this outside the immediate scope of this work.

   (b) "*Bring To Front*". This method takes all the re-incident nodes and brings them to the front of their sibling choice-boxes (or and-continuations). It can be very efficient for some rather poorly-written programs, but it easily leads to bad results because of its "randomizing" effect, as we verified experimentally. This is also the simplest method to implement efficiently.

   (c) "*Incremental*". Instead of bringing the re-incident nodes to the front it moves them to the left by one place. The intuition behind this method is that it will incrementally approximate the place where it should go to as defined by (3a), until it *passes over* it, after which calls to the goal will suspend, waiting for the producer goal to effect its binding. This method should be more "progressive" than method (3b), and will hopefully provide a better approximation of a monotone gain in performance.

4 Statistics

We have implemented the method, with variuos reordering strategies, in the AKL 0.0 prototype implementation developped at SICS [7]. This implementation is not as efficient as state-of-the-art Prolog compilers such as Sicstus Prolog, being 3 to 4 times slower. We believe however that the results reported here can be extrapolated to a more efficient implementation as the method is quite independent of the inference engine. For this purpose, we give as measurement of a reordering strategy's performance the number of choice splits (ie. nondeterminate promotions) performed, as this is probably the most expensive operation in AKL. It corresponds to choice-point creation in normal Prolog. This numbers are thus independent of the underlying inference engine raw speed. The results for mesurements in cpu time and in overall number of inferences are similar to those in number of choice split, and are not included for this reason.

The overhead introduced by the need to maintain the structures necessary for the reordering method results in a slowdown in the order of 5 to 10% over the un-modified AKL implementation, depending on the programs.

Table 1 depicts the relative performance of the various methods we described previously, the labels on the columns stand for:
- `btf`. Bring to front. Reorder when: (1a). What: (2a). Where to: (3b).
- `btf/gp`. Bring to front, same grandparent. Reorder when: (1a) and (1b). What: (2a). Where to: (3b).
- `btf/t`. Bring to front, tree. Reorder when: (1a). What: (2b). Where to: (3b).
- `sbo`. Step by one. Reorder when: (1a). What: (2a). Where to: (3c).
- `sbo/gp`. Step by one, same grandparent. Reorder when: (1a) and (1b). What: (2a). Where to: (3c).
- `sbo/t`. Step by one, tree. Reorder when: (1a). What: (2b). Where to: (3c).

Table 1 displays the performance of these methods relative to the un-modified AKL strategy, as a ratio where numbers greater than 1 represent a speedup (ie. a lower number of choice split operations).

	Strategy					
Benchmark	btf	btf/gp	btf/t	sbo	sbo/gp	sbo/t
circuit	1.38	1.38	1.38	1.00	1.00	1.00
bagof-circuit	2.91	2.91	2.91	1.19	1.28	1.52
color-13-good	0.56	0.68	0.56	0.26	0.91	0.27
color-13-bad	0.74	0.97	0.74	0.34	1.71	0.46
example-g	2.75	2.75	2.75	2.75	2.75	2.75
crypt	1.00	1.00	1.00	1.00	1.00	1.00
knights-5	0.97	1.00	0.97	0.97	1.00	0.97
queens-8	2.09	1.53	2.09	1.05	1.00	1.05
ham	0.90	1.00	0.95	1.00	1.00	1.00
blocks	–	1.20	–	1.97	1.00	1.97
money	1.00	1.00	1.00	1.00	1.00	1.00
bagof-salt-mustard	1.86	1.00	2.01	1.20	1.00	1.33
zebra	2.82	2.54	2.82	1.84	1.80	1.78

Table 1: ratios of Number of Choice Splits (unmodified/modified AKL)

`bagof-circuit` generates all solutions to a digital circuit fault diagnosis problem from [11], `zebra` is a version of the "houses" problem, `crypt` is adapted from [17], `money` is the "send+more=money" crypt-arithmetic puzzle, `example-g` is the example previously discussed, `bagof-salt-mustard` is the "salt-and-mustard" puzzle, `color-13-good` and `color-13-bad` are the traditional map-coloring puzzle from [1] with both a favorable and a bad goal ordering, `ham` and `bagof-ham` consist in finding a Hamiltonian path through a graph, and `blocks` is a simple planning problem in the blocks-world.

The overall best method seems to be the `btf` family, the fastest ones being `btf` and `btf/t`, as they obtain an average 60% gain in performance on these benchmarks, with peak speedups up to a factor three. Some remarks concerning the strategies on this data:

1. The methods that reorder the entire sub-tree leading to the re-incident nodes (names ending in `/t`) perform almost identically to those that only reorder the re-incident nodes. This may be attributable to the fact that the and-or trees represented by the AKL configurations encountered during these benchmarks are not very deep.

2. The "incremental" method (`sbo`) is more stable than the "bring to front" (`btf`) one. In practice this means that the gains are not as high, however, there are not as many pathological cases.

3. The "same grandparent" restricting heuristic (1b, names ending in `/gp`) seems to restrict the gains a little bit, however and most interestingly it prevents some of the truly pathological behaviour of the strategy it's being used with.

4. The `blocks` benchmark with the `btf` and `btf/t` heuristics took too long to complete so the figures are not in the table.

5 Conclusion and further work

We have shown that it is feasible to apply simple search-space pruning heuristics to improve the non-deterministic behaviour in the Andorra-based concurrent logic Language. The experimental results are encouraging and the methods we have described are susceptible of being implemented efficiently and with few modifications to the original engine.

Hence we believe that *a priori* pruning, such as the one provided by the Andorra principle, should be combined with a posteriori pruning, based on failure information, to achieve the best results.

These results can be immediately applied to automated deduction applications, which are both in demand for non-determinism and for control of the search space, as proposed by nondeterministic concurrent logic languages. This is an example of the synergy between research on parallel implementations (concurrent logic languages) research on deduction systems (reordering method based on failure).

References

[1] M. Bruynooghe and L. M. Pereira. Deduction revision by intelligent backtracking. In J.A. Campbell, editor, *implementations of Prolog*. Ellis-Horwood, 1984.

[2] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the basic andorra model. In Furukawa [4], pages 825–839.

[3] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I preprocessor: Supporting full Prolog on the basic andorra model. In Furukawa [4], pages 443–456.

[4] Koichi Furukawa, editor. *Proceedings of the Eighth International Conference on Logic Programming*, Cambridge, Massachusetts London, England, 1991. MIT Press.

[5] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its Computation Model. In Warren and Szeredi [16], pages 31–46.

[6] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra kernel language. In Saraswat and Ueda [13], pages 167–186.

[7] Sverker Janson and Johan Montelius. Design of a sequential prototype implementation of the andorra kernel language. Technical report, SICS, September 1991. (draft).

[8] Robert A. Kowalski and Kenneth A. Bowen, editors. *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Cambridge, Massachusetts London, England, 1988. MIT Press.

[9] Jean-Louis Lassez, editor. *Proceedings of the Fourth International Conference on Logic Pro gramming*, MIT Press Series in Logic Programming, Cambridge, Massachusetts London, England, 1987. "MIT Press".

[10] Giorgio Levi and Maurizio Martelli, editors. *Proceedings of the Sixth International Conference on Logic Programming*, Cambridge, Massachusetts London, England, 1989. MIT Press.

[11] S. Morishita, M. Numao, and S. Hirose. Symbolical construction of truth value domain for logic program. In Lassez [9], pages 533–555.

[12] Lee Naish. Heterogeneous SLD resolution. *The Journal of Logic Programming*, 1(4):297–303, December 1984.

[13] Vijay Saraswat and Kazunori Ueda, editors. *Logic Programming Proceedings of the 1991 International Symposium*, London, England, 1991. Massachusetts Institute of Technology.

[14] Ehud Shapiro, editor. *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1986.

[15] D. H. D. Warren. The andorra principle. Internal report, Gigalips Group, 1988.

[16] David H. D. Warren and Peter Szeredi, editors. *Proceedings of the Seventh International Conference on Logic Programming*, Cambridge, Massachusetts London, England, 1990. MIT Press.

[17] R. Yang. Solving simple substitution ciphers in Andorra-I. In Levi and Martelli [10], pages 113–128.