Improving Backward Execution in the Andorra Family of Languages

Salvador Abreu
Luís Moniz Pereira
CRIA Uninova and DCS, 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 reschedule 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. We have implemented this scheme by modifying the Andorra Kernel Language prototype implementation developed at SICS (AKL/PS 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 of Prolog. On one hand, the depth-first left-to-right strategy of Prolog, due to its simple stack-based execution model, has led to tremendous advances in Prolog implementation technology which has been decisive for establishing logic programming as a mature declarative paradigm. But on the other hand, one sometimes prefers 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. Moreover, a few implementations of concurrent languages now achieve raw speeds close to that of traditional Prolog systems, cf. [8, 14, 9].

Among all the concurrent logic languages, see [22] for a complete presentation and a detailed history of this programming paradigm, the Andorra family of languages is one of the most promising lines of research in this quest to tame parallelism and concurrency, as they try to encompass both Prolog and concurrent language programming styles and to take the best of both worlds. The *Andorra model* was proposed by D.H.D. Warren [25] 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 [6, 5] and the Andorra Kernel Language[11, 12]. The essential idea is to execute determinate goals first and concurrently, delaying the execution of non-determinate goals until no determinate goal can proceed. This was inspired by the design of the concurrent language P-Prolog[29] where synchronization between goals was based on the concept of determinacy of guard systems. However the roots of such a concept can be traced back further to the early developments of Prolog, as for instance in the *sidetracking* search procedure of [20] which favors the development of goals with the least alternatives. This is indeed but another instance of the "first-fail" heuristic often used in various fields of problem solving, that has for instance recently shown its usefulness in the area of Constraint Logic Programming [24].

An interesting aspect of the Andorra principle, is the ability to reduce the size of the computation when compared to standard Prolog, as early execution of determinate goals can amount to an *a priori* pruning of the search space. [5] shows that the reduction of the number of inferences of Andorra w.r.t. Prolog can attain one order of magnitude.

Although much work has been done to improve the forward execution in logic languages, as exemplified by the Andorra model, very little works deals with the improvement of the backward execution of programs. It is 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, if the backtrack point is not pertinent. This goal should thus be tried sooner after backtracking, as this could immediately amount to further deep 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 applied 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[19] 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 another, the whole computation tree has to be explicitly represented.

One important aspect of our work is that the information we use for the backward execution is very limited and the goal of this work is *not* to attempt anything like intelligent backtracking as has been proposed for Prolog [1, 3, 4] or concurrent languages [2]. Intelligent backtracking would require using data-dependency information in some way. This requires to record more information, usually during the unification process, in order to determine dependencies between failing goals and choice-points, based on data-dependencies. We preferred to experiment to what extent we could have an efficient strategy, which would behave most of the time like an intelligent backtracking scheme, without the need for data-dependency information. Also observe that in an Andorra-based model, some of the "intelligence" is already provided by the forward execution mechanism in the delaying of choices, and the eager execution of related determinate goals after the creation of a choice-point. Hence intelligent backtracking is interesting as a cure for the rigid and naive strategy of a language like Prolog, but it seems less necessary in a concurrent language as Andorra, at least when only transformational and not reactive computations are considered. We are thus only using the "fail information" as a heuristic to guide the process of pruning the search space. We shall concentrate on computationally simple strategies, guided as much as possible by information already available, this will be detailed in section 3.

We have implemented this scheme in the prototype version of the Andorra Kernel Language (AKL 0.0) developed at SICS [13]. As mentioned above, one of the main interests 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 rescheduling 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 60% (as measured by the number of nondeterminate promotions) can be observed, with peak values up to 300%.

The rest of this article is structured as follows: section 2 summarizes the Andorra Kernel Language, and we focus on the issues most relevant for our work. Then, in section 3 we present the various methods we have experimented with. We detail some implementations issues in section 4. Section 5 illustrates this with an example. We then proceed in section 6 to present and analyze some statistics on the observed performance of the prototype implementation. Finally, in section 7 some conclusions are drawn and our projects for further developments are outlined.

2 Andorra Kernel Language

The Andorra Kernel Language (AKL) has been proposed in [12] and is a simplified instance of the framework presented in [11]. It is an attempt to fully encompass both Prolog and concurrent logic languages (aka committed-choice languages) such as Parlog, FCP or GHC [22]. It can be seen in this way as an instance of the Extended Andorra Model where the control is explicit, as opposed to [26] where the emphasis is put on an implicit control.

The main new language feature is the introduction of the guard construct borrowed from the concurrent logic programming paradigm, and its associated guard operator. However, both don't care and don't know non determinism are possible by using as guard operators either "commit" and "cut" (which prune alternatives), or "wait" (which does not prune alternatives). Guard atoms are not restricted to build-in predicates but can be program atoms, leading to non flat guards. The introduction of guards in the program's clauses is indeed a way to extend the determinacy test, which is at the core of the Andorra model, since a goal is determinate whenever a single guard check succeeds among all alternative clauses, and not only a single head unification.

AKL is a practical instance of the constraint-based framework of

[11] with three possible guard operators (cut, commit and wait) and the constraint operation tell_ω on unification constraints. A further restriction consists in allowing quiet pruning only, meaning that a pruning guard operator (cut or commit) may only be used if the head and the guard make no non-local bindings.

The semantics of the language is given by a set of rewrite rules applicable to and-or trees designated as *configurations*. Briefly the process of solving a goal G w.r.t. a given AKL program P can be formalized as finding a sequence of configurations C_i, starting with the configuration $C_0 = G$, with successive configurations obtained by application of the rewriting rules (see 2). In the simplified description that follows we omit binding environments (constraints), and consider the simplified AKL execution model, ie. without or-boxes[13].

Configurations

A configuration models the state of the computation. It can briefly be summarized by the following (very) simplified grammar:

$$\begin{array}{rcl} \mathit{Configuration} & \longrightarrow & \mathit{Goal} | \mathbf{choice}(\mathit{list}(\mathit{And\_Box})) \\ \mathit{And\_Box} & \longrightarrow & \mathbf{fail} | \mathbf{and}(\mathit{list}(\mathit{Configuration})) \\ \mathit{Goal} & \longrightarrow & \mathit{Lits}_{\mathit{Guard}} \% \mathit{Lits}_{\mathit{Body}} \end{array}$$

Where *Lits* stands for a conjunction of literals, and % is a guard operator, namely wait (:), cut (!) or commit (|).

Rewriting Rules

Again these are a simplified version of the rewriting rules, making no mention of the binding environments. The rewrite rules apply to either choice-boxes or and-boxes, which may be contained in other boxes.

1. **Local Forking**. Given a subgoal A and considering that the predicate designated by A has the clauses $H_1 \leftarrow G_1 \% B_1, \ldots, H_n \leftarrow G_n \% B_n$, where % is a guard operator:

$$A \Longrightarrow \mathbf{choice}(\mathbf{and}(G_1)\%B_1, \ldots, \mathbf{and}(G_n)\%B_n)$$

2. **Failure Propagation**.

$$\mathbf{and}(\ldots, \mathbf{fail}, \ldots) \Longrightarrow \mathbf{fail}$$

3. **Choice Elimination**.

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

4. **Environment Synchronization.**

$$\mathbf{and}(R) \Longrightarrow \mathbf{fail}$$

iff R's constraints are incompatible with the and-box's bindings. Basically this is unification failure.

5. **Determinate Promotion.**

$$\mathbf{and}(R, \mathbf{choice}(C\%A), S) \Longrightarrow \mathbf{and}(R, C, A, S)$$

If % is a pruning guard operator (ie. cut or commit) it must also be quiet.

6. **Cut.**

$$\mathbf{choice}(R, C!A, S) \Longrightarrow \mathbf{choice}(R, C!A)$$

7. **Commit.**

$$\mathbf{choice}(R, C|A, S) \Longrightarrow \mathbf{choice}(C|A)$$

8. **Choice Splitting.**

$$\begin{array}{l}\mathbf{choice}(R_1, \mathbf{and}(S_1, \mathbf{choice}(T_1, G : B, T_2), S_2), R_2) \Longrightarrow \\ \quad \mathbf{choice}(R_1, \mathbf{and}(S_1, \mathbf{choice}(G : B), S_2), \mathbf{and}(S_1, \mathbf{choice}(T_1, T_2), S_2), R_2)\end{array}$$

For a more extensive description, see [12] and [13].

Copying Scheme for handling Nondeterminism

The current (sequential) implementation of AKL [13] uses copying for the *choice splitting* operation (8), as described by the semantic rewrite rule. This operation is depicted in figure 1: (a) shows the configuration before applying rule (8), and (b) afterwards.

This operation leads to a very different run-time organization from that of 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 can be seen as an eager generation of alternative resolvents, as opposed to the lazy policy of using choice-points in traditional Prolog. Surprisingly enough, this scheme is related to the proof procedure of [7] where alternative resolvents are also generated eagerly.

In (b), choice-boxes S1' and S2' are *copies* of S1 and S2. This copying scheme on choice split will greatly influence the operational definition of our backward execution strategy. Indeed, our aim of reordering continuation goals upon failure so as to re-execute first the

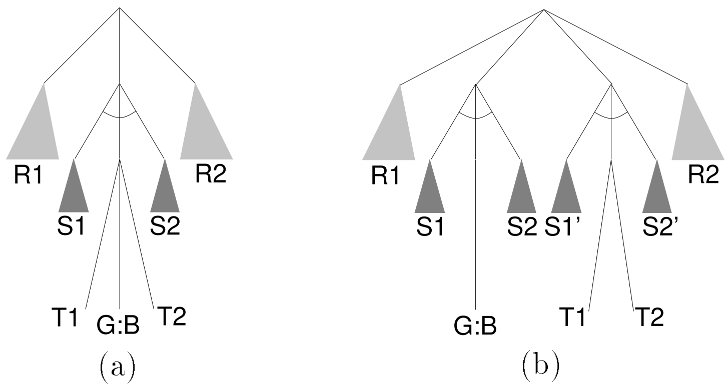

Figure 1: Configurations before and after choice split.

failing goal has to be rephrased as a transformation and reordering of the and-or tree upon failure. In particular, a goal in a given and-box has to be connected to its other occurrences in alternative ("copied") and-boxes in such a manner that that these other occurrences can be easily reordered upon failure. 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

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.

In any case, reordering only takes place upon occurrence of a failure and basically consists in moving re-incident nodes of the failing goal so that they will be executed earlier that they normally would with the standard (non-reordering) strategy. This corresponds roughly to moving these nodes to the left of the tree.

Reordering strategies

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

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.* It is reasonable to impose this condition as a call to a predicate may succeed although the first clauses fail. This corresponds to reorder upon deep backtracking only.

   (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. Nodes in a copy chain may be at different depths in a given configuration as a result of deterministic promotions, where the and-box being promoted carries with it its parent choice-box's copy information into the newly created and-continuation.

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.

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) "*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.

(b) "*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 producer of the conflicting binding, 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 (3a), and will hopefully provide a better approximation of a monotone gain in performance.

4 Implementation Issues

The characteristic that all of these methods have in common is their relative simplicity of implementation, and the fact that they require only small extensions to the data structures of the abstract machine emulator.

In particular, there is no modification to the representation of terms or variables. The changes consisted in modifying the structure of and-boxes, choice-boxes and and-continuations so as to make them appear as subclasses of a "copiable object", that will provide a link to other "copiable objects", as well as some information on the type of the object. Copy chains are maintained that link all the re-incident nodes of a given node. The nodes involved in the copy chains are always either choice-boxes or and-continuations.

The operations that incur an increased overhead are:

- Failure. With this scheme an and-box that is being discarded must be traversed in order to remove all the nodes below it from their copy chain. Ideally this operation should be deferred until garbage collection.
- Tree copying that occurs during choice split operations. All nodes that get copied must now be linked together.
- Tree copying that occurs during garbage collection. This operation now has to relocate a structure which is independent of the tree, the copy chains. A fix-up stack has to be maintained.

The overhead introduced by the need to maintain the structures necessary for the node-copy information results in a slowdown in the order of 5 to 10% over the un-modified AKL implementation.

5 Example Execution

We will detail the AKL execution mechanism and the effect of the reordering strategy upon failure with the simple example program:

```
p(X) :- : X=1.
p(X) :- : X=2.  q(X) :- : X=a.
                q(X) :- : X=b.
r(X) :- : X=0.  q(X) :- : X=c.
r(X) :- : X=2.
```

An AKL configuration corresponding to the execution of goal `:- p(X), q(Y), r(X)` is shown in part (a) of figure 2, before any nondeterministic step (ie. choice splitting) is made. At this stage, a choice splitting is the only applicable rule and it is literal `p` that is chosen, with its first alternative being then selected for determinate promotion. This yields the configurations shown in parts (b) and (c) of figure 2, with `p` reduced to an and-continuation.

As the nodes labeled q' and r' are copies (re-incident nodes) of those labeled q and r, the abstract machine emulator will make this fact explicit by establishing links from the original node to its copies, hence the arrows that link nodes horizontally. The direction of the arrow is not very important as the lists are circular, but an arrow $A \longmapsto B$ means that B is a (potential) re-incident node of A. The dotted lines stand for *and-continuations* that represent the yet to be executed prefix of the resolvent corresponding to the remainder of the body of a given clause.

Normal AKL execution will proceed by successive application of the rewriting rules. Whenever a goal fails, that is when rules (4) or (2) are applied, the tree reordering mechanism will come into play, relocating the other instances of the goal that failed (ie. the re-incident nodes), using the strategy currently in effect, as specified by (3).

Continuing with the example, this can be seen on the tree in part (h) of figure 2. In this case the criterion used was `sbo/gp` (see table 1).

6 Statistics

We have implemented the method, with various reordering strategies, on top of the AKL 0.0 prototype implementation developed at SICS[13]. 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 measurements in cpu time and in overall

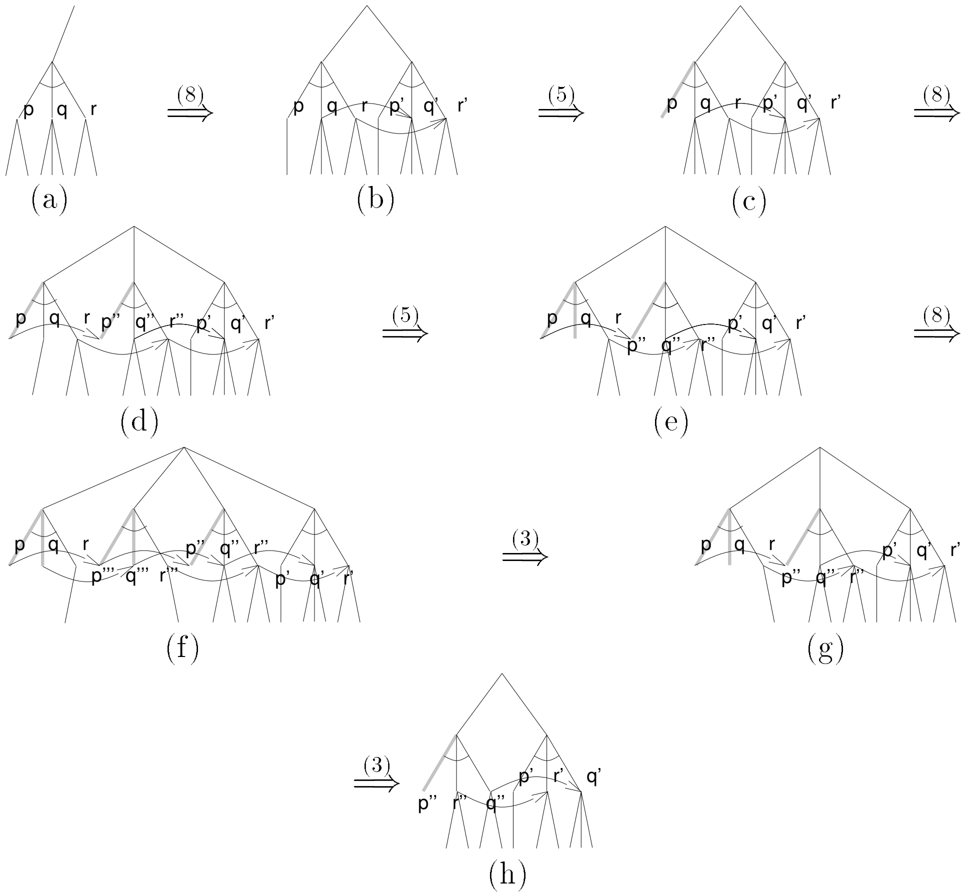

Figure 2: Configurations for `:- p(X), q(Y), r(X)`

number of inferences are similar to those in number of choice split, and are not included for this reason.

Table 2 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). Table 1 describes the strategies.

`bagof-circuit` generates all solutions to a digital circuit fault diagnosis problem from [18], `zebra` is a version of the "houses" problem, `crypt` is adapted from [28], `money` is the "send+more=money" cryptarithmetic puzzle, `example-g` is the example discussed in section 5, `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

Label	Strategy name	Reorder: when	what	where to
`btf`	Bring to front	(1a)	(2a)	(3a)
`btf/gp`	Bring to front, same grandparent	(1a), (1b)	(2a)	(3a)
`btf/t`	Bring to front, tree	(1a)	(2b)	(3a)
`sbo`	Step by one	(1a)	(2a)	(3b)
`sbo/gp`	Step by one, same grandparent	(1a), (1b)	(2a)	(3b)
`sbo/t`	Step by one, tree	(1a)	(2b)	(3b)

Table 1: Strategy summary

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

performance on these benchmarks, with peak speedups up to a factor three. Some remarks concerning the strategies on this data:

- The methods that reorder the entire sub-tree leading to the re-incident nodes (`*/t`) perform almost identically to those that only reorder the re-incident nodes. This may be due to the AKL configurations being shallow.
- 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.
- The "same grandparent" restricting heuristic (1b, `*/gp`) seems to restrict the gains a little, however it prevents the pathological behavior of the strategy it's being used with to appear.

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

Concerning the effectiveness of the reordering strategy in general, we can make the following remarks:

- The method (all strategies taken together) is ineffective for some programs, as depicted by lines of 1.00's (or very close values) for the `crypt`, `knights`, `money`, `bagof-salt-mustard` and `ham` programs. This is because these programs are deterministic in the sense that they never require deep backtracking or are "fully" non-deterministic, in the sense that all choice points have to be explored. In such situations, where any backtracking is always pertinent, the reordering method could obviously not improve performance. However, as soon as there exist testing goals that are pending on more than one non-determinate promotion, our method can significantly improve performance.

- The Andorra Principle reduces the number of inferences needed to solve a problem when compared to Prolog, but failure information on backward execution can further improve this. *A priori* pruning should be combined with a posteriori pruning.

- Some of the non-deterministic examples were adapted[2] from similar ones used for benchmarking intelligent backtracking methods[4, 2]. They do not benefit as much from this type of manipulation as we initially expected; this is due to the fact that the Andorra principle already improves the "naive" version. A striking example of this is the `color-13` program. A good order of goals in the query gives the solution immediately and deterministically, while a bad order of goals needs 2000 times more inferences to give it in Prolog, but in AKL the two goals have roughly the same behaviour (only 1.13 times more inferences).

7 Conclusion and further work

We have shown that it is feasible to apply simple search-space pruning heuristics to improve the nondeterministic behavior in the Andorra Kernel Language. This work applies to the Extended Andorra Model as well. 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.

Re-scheduling based on dependency analysis is another approach, that we also intend to pursue, also assuming the Extended Andorra Model, in order to investigate what we believe is its greater appropriateness than that of the stack-based WAM model to accommodate intelligent backtracking and related notions. Such an approach combined with the Andorra Principle should be more effective in the context of reactive computations than in the purely transformational view, as independent parts of the computation need not be recalculated.

We are also interested in extending those ideas to Constraint Logic Programming, and to concurrent constraint languages in particular, and to use such heuristics to efficiently handle disjunctive constraints.

Acknowledgements

This work has been supported in part by JNICT and MRT, through the "Réseau de formation/recherche", which provided funding for Salvador Abreu's stay at INRIA during which this work was done.

Notes

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

[2]Many of these programs were written in Prolog and had to be hand-translated into AKL. Programs making use of the database could of course not be translated directly.

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] C. Codognet and P. Codognet. Non-deterministic stream AND-Parallelism based on intelligent backtracking. In Levi and Martelli [17], pages 63–79.

[3] Christian Codognet, Philippe Codognet, and Gilberto File. Yet another intelligent backtracking method. In Kowalski and Bowen [15], pages 447–465.

[4] Philippe Codognet and Thierry Sola. Extending the WAM for intelligent backtracking. In Furukawa [10], pages 127–141.

[5] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the basic andorra model. In Furukawa [10], pages 825–839.

[6] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I preprocessor: Supporting full Prolog on the basic andorra model. In Furukawa [10], pages 443–456.

[7] P. T. Cox and T. Pietrzykowski. Deduction plans : a basis for intelligent backtracking. *IEEE PAMI*, 3(1), 1981.

[8] J. Crammond. *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors*. PhD thesis, Department of Computer Science, Herriot-Watt University, Edinburgh, May, 1988.

[9] Ian Foster and Steven Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[10] Koichi Furukawa, editor. *Proceedings of the Eighth International Conference on Logic Programming*, Paris, France, 1991. The MIT Press.

[11] Seif Haridi and Sverker Janson. Kernel andorra Prolog and its computation model. In Warren and Szeredi [27], pages 31–46.

[12] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra kernel language. In Saraswat and Ueda [21], pages 167–186.

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

[14] Shmuel Klinger and Ehud Shapiro. A decision tree compilation algorithm for FCP (I , : , ?). In Kowalski and Bowen [15], pages 1315–1336.

[15] 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.

[16] Jean-Louis Lassez, editor. *Proceedings of the Fourth International Conference on Logic Pro gramming*, MIT Press Series in Logic Programming, Melbourne, 1987. The MIT Press.

[17] Giorgio Levi and Maurizio Martelli, editors. *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon, 1989. The MIT Press.

[18] S. Morishita, M. Numao, and S. Hirose. Symbolical construction of truth value domain for logic program. In Lassez [16], pages 533–555.

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

[20] L. M. Pereira and A. Porto. Intelligent backtracking and sidetracking in horn clause programs. Technical Report CIUNL 2/79, Universitade Nova de Lisboa, 1979.

[21] Vijay Saraswat and Kazunori Ueda, editors. *Logic Programming, Proceedings of the 1991 International Symposium*, San Diego, USA, 1991. The MIT Press.

[22] E. Shapiro. The family of concurrent logic programming languages. *ACM computing surveys*, 21(3), september 1989.

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

[24] P. Van Hentenryck. Parallel constraint satisfaction in logic programming: Preliminary results of chip within PEPSys. In Levi and Martelli [17], pages 165–180.

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

[26] David H. D. Warren. The extended andorra model with implicit control. ICLP90 Preconference Workshop, June 1990.

[27] David H. D. Warren and Peter Szeredi, editors. *Proceedings of the Seventh International Conference on Logic Programming*, Jerusalem, 1990. The MIT Press.

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

[29] Rong Yang and Hideo Aiso. P-prolog: a parallel logic language based on exclusive relation. In Shapiro [23], pages 255–269.