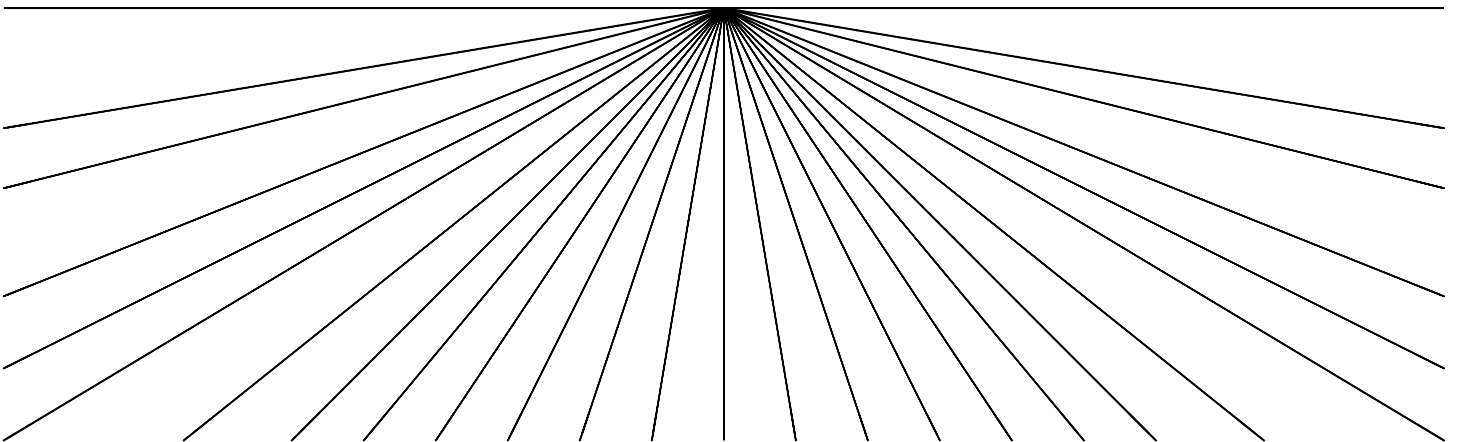
facultad de informática

universidad politécnica de madrid

QE-Andorra: A Quiche–Eating Implementation of the Basic Andorra Model

F. Bueno
S. Debray
M. García de la Banda
M. Hermenegildo

TR Number CLIP13/94.0



QE-Andorra:
A Quiche–Eating Implementation of the Basic Andorra Model

Technical Report Number: CLIP13/94.0

September 1994

Authors

F. Bueno, **M. García de la Banda**, **M. Hermenegildo**
Universidad Politécnica de Madrid (UPM), Facultad de Informática, 28660-Boadilla del Monte, Madrid - Spain,

S. Debray
University of Arizona, Computer Science Department, 85721 Tucson, Arizona, USA

Keywords

Logic Programming, Concurrent Programming, Andorra Model, Program Transformation, Language Implementation

Acknowledgements

We would like to thank D.H.D. Warren, V. Santos-Costa, R. Yang, and the rest of the Andorra-I team for making a version of the Andorra-I system available to us and for useful discussions. This work has been funded in part by ESPRIT project #7195 "ACCLAIM" and by CICYT project TIC93-0975-CE.

Abstract

A recent trend towards convergence in the analysis and implementation techniques for CC and CLP systems can be observed: while the respective implementations and characteristics of these languages are in principle very different, CLP (and Prolog) systems have been incorporating capabilities to deal with user–defined suspension and coroutining behaviors using very similar implementation techniques (and requiring very similar analyses) to those used in CC languages. Conversely, CC compilers have been trying to coalesce fine–grained tasks into coarser–grained sequential threads that are implemented using stack–based techniques similar to those used in sequential languages. This convergence of techniques opens up the possibility of having a general purpose kernel language and abstract machine to serve as a compilation target for a variety of user–level languages.

In this report, we propose a transformation technique, aided by analysis, directed towards the above mentioned objective. In particular, we report on techniques to support the Andorra computational model, essentially emulating the Andorra-I system, via program transformation into a sequential language with delay primitives. The system is easily automatable, comprising a simple program analyzer and a basic transformer to the kernel language. It turns out that a simple (parallel) CLP (or Prolog) system with dynamic scheduling is sufficient as a kernel language for this purpose. The preliminary results are quite encouraging: performance of the resulting system is comparable to the current Andorra-I implementation.

Contents

1 Introduction: Towards General-Purpose Implementations 1

2 A General Transformation for the Andorra Model 4

3 Determinacy Conditions and Suspension Declarations 7

4 Handling Non-Pure Features 8

5 Mixing Prolog and Andorra-I Code 10

6 Optimizing the Transformation 10

6.1 Simple enhancements 11

6.2 Determinacy condition is true 11

6.3 Determinacy condition is false 12

6.4 Unchaining calls to predicates 12

6.5 An optimized algorithm 13

7 An Example 15

8 Performance Figures 16

9 Conclusions 17

References 19

1 Introduction: Towards General-Purpose Implementations

Many current proposals for parallel or concurrent logic programming languages and models are actually "bundled packages", in the sense that they offer a combined solution affecting a number of issues such as choice of computation rule, concurrency, exploitation of parallelism, etc. This is understandable since a practical model has to offer solutions for all the problems involved. However, the bundled nature of (the description of) many models often makes it difficult to compare them with each other. As a result, implementors typically build their systems from scratch, by writing their own runtime systems and constructing compilers to compile programs into low-level languages such as C or assembler.

The tremendous engineering and manpower overheads involved in such an enterprise means that in many cases, implementors may be unable to take advantage of clever optimizations that other researchers have implemented, or to invest the time and effort necessary to turn their systems from research prototypes to mature and robust systems that can be shared with other researchers and users. Some researchers find the "non-research" engineering overhead sufficiently daunting that their ideas and languages do not make it past the paper design stage.

This is an unfortunate state of affairs, and leads to a great deal of duplicated effort and wasted time. It is argued in [16] that this situation can be improved by performing a "separation analysis" of the parallel or concurrent model underlying the language and isolating its fundamental principles. On the one hand, such un-bundling shows that the applicability of these fundamental principles can be enlarged by allowing the transference of the valuable features of a model to another. On the other hand, the study reveals the existence of several fundamental principles which are common to several models. This fact at the same time explains and is supported by the recent trend towards convergence in the analysis and implementation techniques of models that are in principle very different, such as the various parallel implementations of Prolog on one hand (see, for example, [15, 24, 27]) and the implementations of the various committed choice languages on the other (see, for example, [4, 10, 19, 32, 33]).

The aforementioned convergence of parallel and concurrent systems can be observed in that, on one hand, driven by the demonstrated utility of delay primitives in sequential Prolog systems (e.g., the `freeze` and `block` declarations of Sicstus Prolog [3], `when` declarations of NU-Prolog [30], etc.), parallel Prolog systems have been incorporating capabilities to deal with user-defined suspension and coroutining behaviors. In sequential Prolog systems with delay primitives, delayed goals are typically represented via heap-allocated "suspension records," and such goals are awakened when the variables they are suspended on get bindings [2]. Parallel Prolog systems inherit this architecture, leading to implementations where individual tasks are stack-oriented, together with support for heap-allocated suspensions and dataflow synchronization. For example, &-Prolog allows programmer-supplied *wait*-declarations, which can be used to express arbitrary control dependencies, in addition to the more standard wait declarations. On the other hand, driven by a growing consensus that some form of "sequentialization" is necessary to reduce the overhead of managing fine-grained parallel tasks on stock hardware (see, for example, [9, 31, 22, 11]), implementors of committed choice languages are investigating the use of compile-time analyses to coalesce fine-grained tasks into coarser-grained sequential threads that can be implemented more efficiently. This, again, leads to implementations where individual sequential threads execute in a stack-oriented manner, but where sets of such threads are represented via heap-allocated activation records that employ dataflow synchronization. Interestingly, and conversely, in the context of paral-

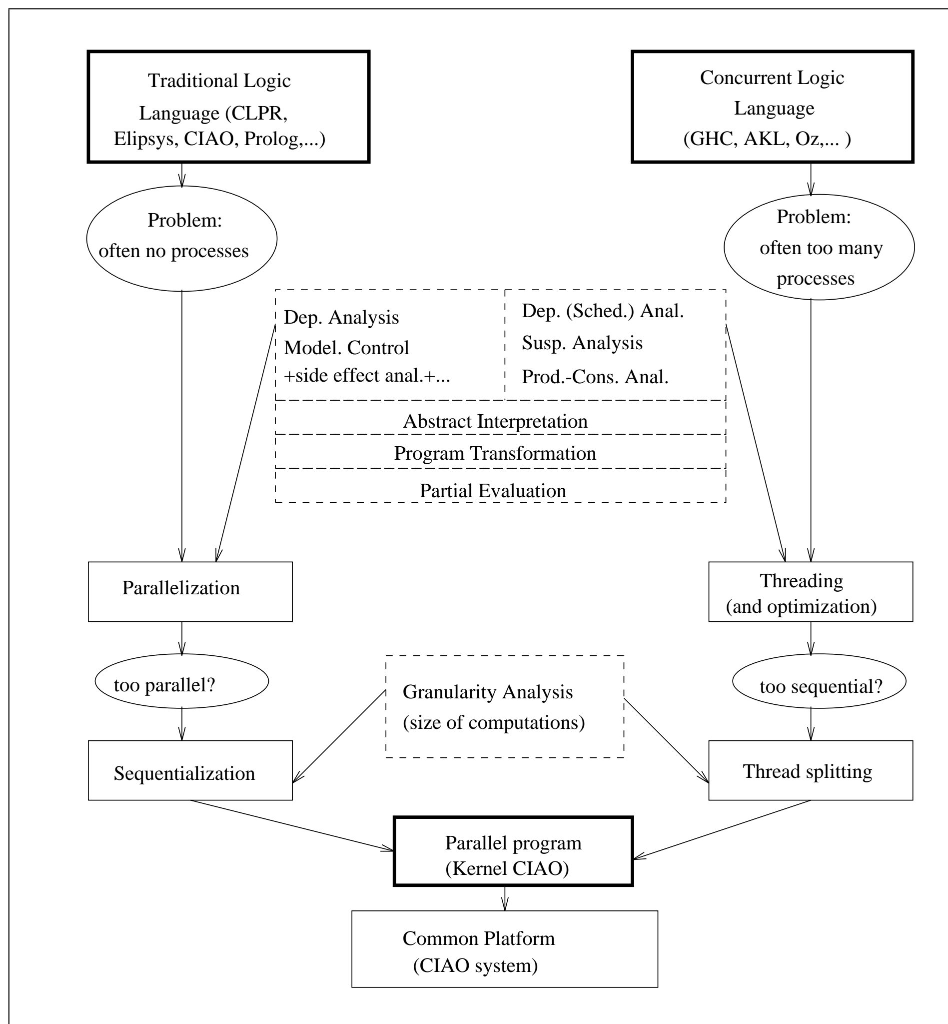


Figure 1: Common Problems in Compilation

lel Prolog systems, there is also a growing body of work trying to address the problem of automatic parallelizing compilers often "parallelizing too much" which appears if the target architecture is not capable of supporting fine grain parallelism. Figure 1 illustrates this and underlines the commonality of techniques at the compiler level (a similar parallel could be drawn at the implementation level).

This convergence of trends both at the compiler and the run-time system levels is exciting: it suggests that we are beginning to understand the essential implementation issues for these languages, and that from an implementor's perspective these languages are not as fundamentally different as was originally believed. It also opens up the possibility of having a general purpose kernel language and abstract machine that supports the features needed by various parallel logic programming languages. Given a sufficiently high level intermediate language of this kind, and carefully crafted compilers and runtime systems for this language, the implementation of other logic programming languages would be simplified considerably: rather than reinvent all aspects of an implementation from scratch, it would be possible to share the "back end" across different systems, and thus require only the construction of compilers to the intermediate representation. In summary, it opens

up the possibility of having a general purpose kernel language and abstract machine to serve as a compilation target for a variety of user-level languages.

Encouraging initial results in this direction have been demonstrated in the sequential context by the QD-Janus system [8] of S. Debray and his group. QD-Janus, which compiles down to Sicstus Prolog and uses the delay primitives of the Prolog system to implement dataflow synchronization, turns out to be more than three times faster, on the average, than Kliger's customized implementation of FCP(:) [23] and requires two orders of magnitude less heap memory [7]. We believe that this point will also extend to other logical languages and to parallel systems, given a suitable kernel language.

We propose to use the *Ciao* (Concurrent/Constraint Independent And-Or parallel) model as such general purpose kernel system. *Ciao* [16] can be viewed as an and generalization of the &-Prolog system, thus inheriting a stack-oriented parallel execution together with arbitrary control dependencies, suspension, and data-flow synchronization via user-supplied *wait*-declarations. This suggests that, with some enhancements, some of which will be mentioned below, the dependency graphs and *wait*-declarations of &-Prolog/*Ciao*, can serve as a common intermediate language, and its runtime system can act as an appropriate common low-level implementation, for a variety of parallel logic programming implementations.

Among other things, *Ciao* adds to &-prolog the possibility of attaching attributes to variables [20, 2]. Incorporating the possibility of attaching attributes to variables in a logic programming system has been shown to allow the addition of general constraint solving capabilities to it [17, 18]. This approach is very attractive in that by adding a few primitives any logic programming system can be turned into a generic constraint logic programming system in which constraint solving can be user defined, and at source level – an extreme example of the "glass box" approach. Thanks to this technique *Ciao* can already be viewed as a generic constraint logic programming system.

Furthermore, as shown in [14], a system which implements attributed variables and a few additional primitives (such as those present in *Ciao*) can be easily customized at source level to implement many of the languages and execution models of parallelism and concurrency currently proposed, in both shared memory and distributed systems. Our solution provides the same "glass box" flavor and user accessibility to the implementation in a generic parallel/concurrent (constraint) logic programming system. We do not mean to suggest that the performance of such a system will be *optimal* for all possible logic programming languages: our claim is rather that it will provide a way to researchers in the community implement their languages with considerably less effort than has been possible to date, and yet attain reasonably good performance. We are currently exploring these points in collaboration with S. Debray, F. Rossi, and U. Montanari, by using the *Ciao* system as a generic implementation platform.

As one of the first steps in our construction, in this paper we show how the *Ciao*, and, in fact, even a standard Prolog/CLP system with delay primitives, can serve as a target language for compiling Andorra, and in particular Andorra-I, programs. We will show how to translate such programs and the kind of optimizations that can be applied both during and after such translation. This transformation will be illustrated by means of a detailed example in which we will follow all the different steps applicable. Finally, we provide some experimental results which are encouraging: performance of the resulting system is comparable to the native Andorra-I implementation [28]. Again, we do not mean to suggest that the performance of a system implemented using our approach is optimal or that it will achieve better results than the native Andorra-I implementation, but rather

that the technique is practical and allows the support of the Basic Andorra Model on a generic system with reasonable performance.

2 A General Transformation for the Andorra Model

As mentioned in the introduction, we would like to transform an Andorra-I program into our kernel language in such a way that the execution of the transformed program follows the execution model implemented by the sequential Andorra-I system, i.e., the Basic Andorra Model. In this model a goal, or more precisely a reduction, is delayed until either (a) it becomes determinate or (b) it becomes the leftmost reduction and no determinate reduction is available. Simulating (a) in Prolog is conceptually simple, although might require complex machinery. Simulating (b) is more involved due to the need of keeping track of the order in which the goals would have been reduced in a sequential system with left-to-right computation rule.

For a program P, this operational semantics can be presented as a transition system on *states* $\langle G, c, D, Inf \rangle$ where G is a sequence of literals, c is a constraint, D is a sequence of delayed literals, and Inf is a given structure. Intuitively, G is the sequence of literals being considered for execution, c is the current store, D contains the sequence of literals which are satisfy neither (a) nor (b), and Inf contains the information necessary for detecting condition (b). The transition system is parameterized by three functions, namely *determinate*, $leftmost$, and add_info. The function $determinate(l, c)$ holds iff the literal l is determinate in the context of constraint c. The function $leftmost(l, Inf)$ holds if, according to Inf, the literal l is leftmost. The function $add_info(l, Inf)$ updates the structure Inf to take into account that the literal l is delayed. Also, and for the sake of simplicity, the system will be also parameterized by the procedure $reduce(\langle G, c, D \rangle)$ which obtains a sub-state $\langle G', c', D' \rangle$ by performing a reduction step from sub-state $\langle G, c, D \rangle$, based on the particular operational semantics of the kernel language. The transitions in the transition system are:

(a) $\langle G, c, D \cup l, Inf \rangle \rightarrow_{wdet} \langle G', c', D', Inf \rangle$ if $determinate(l, c)$ holds and $reduce(\langle l : G, c, D \rangle) = \langle G', c', D' \rangle$.

(b) $\langle l : G, c, D, Inf \rangle \rightarrow_{det} \langle G', c', D', Inf \rangle$ if $determinate(l, c)$ holds and $reduce(\langle l : G, c, D \rangle) = \langle G', c', D' \rangle$.

(c) $\langle G, c, D \cup l, Inf \rangle \rightarrow_{wleft} \langle G', c', D', Inf \rangle$ if $leftmost(l, Inf)$ holds and $reduce(\langle l : G, c, D \rangle) = \langle G', c', D' \rangle$.

(d) $\langle l : G, c, D, Inf \rangle \rightarrow_{del} \langle G, c, l : D, Inf' \rangle$ where $add_info(l, Inf) = Inf'$.

Then, the Basic Andorra model would be simulated by ensuring that transition (d) is applied only if neither (a) nor (b) nor (c) can be applied, and transition (c) is applied only if neither (a) nor (b) can be applied.

There are many ways in which the parameterized functions *determinate* and $leftmost$ can be defined, mainly depending on the concept of *determinacy* chosen and the kind of structure Inf defined, respectively. The concept of determinacy will be instantiated later on in Section 3. Regarding the structure Inf, we will define it as a list of variables, each of them attached to the goal

that would have been reduced in a sequential system with left-to-right computation rule, the relative order among the variables reflecting the relative order among such reductions. In this context, $leftmost(l, Inf)$ will hold if the variable in Inf attached to l has become non variable.

We will build Inf by means of difference lists. In particular, for each program predicate `p/n`, we will create two predicates: `p/n+2` and `p_susp/n+2`. The definition of `p/n+2` is the following:

```
p(X̄,L,L1):-
      L = [S|L0],
      det_susp((det_cond(X̄);nonvar(S)),p_susp(X̄,L0,L1)).
```

where X̄ is a sequence of n distinct variables, for each constraint c, *det_cond*(X̄) holds for c iff *determinate*(p(X̄), c) holds, and *det_susp*/2 is a suspension primitive of the language which delays the goal provided as second argument until the condition provided as first argument becomes true.

Intuitively, `S` is the variable attached to `p(X̄)` and `L0,L1` are the pointers which delimit the sublist in which the variables attached to the subgoals in the derivation tree of `p(X̄,L0,L1)` will appear. Intuitively, Inf can be considered a chain, and `L0` and `L1` can be considered the links which connect the reductions needed to execute `p(X̄,L0,L1)` with those needed to execute the goals to the left and right of `p(X̄,L0,L1)`, respectively.

The definition of `p_susp/n+2` is derived from the definition of `p/n` as follows:

- For every fact `p(X̄).`, we create the fact `p_susp(X̄,L,L)`.
- For every clause `p(X̄):- q1(Ȳ1) ··· qn(Ȳn).`, with $n > 0$, we create the clause `p_susp(X̄,L1,Ln+1):- q1(Ȳ1,L1,L2) ··· qn(Ȳn,Ln,Ln+1).`

When a fact is selected during the execution of `p/n+2`, no further reductions are needed, and therefore we must unify the pointers associated to the goal thus closing the associated list. Otherwise, we need to split up such list in as many sublist as goals appear in the body of the clause, always keeping the left-to-right order.

We have already mentioned that $leftmost(l, Inf)$ holds if the variable in Inf attached to l has become non variable — thus, we will call such variables (`S` in the above definition) the "leftmost–tokens." The instantiation of these tokens is achieved by means of a transformed query.

- For a given query `:- q1(Ȳ1) ··· qn(Ȳn).`, where $n > 0$, we create the new query `:- q1(Ȳ1,L1,L2) ··· qn(Ȳn,Ln,Ln+1),wakeup(L1,Ln+1).`

where `wakeup/2` is defined as follows:

```
wakeup(L1,L2) :- L1==L2, !.
wakeup([L1|L2],L3) :- L1=up, wakeup(L2,L3).
```

With such a translation, and given the operational semantics defined above, it is clear that the sequential execution of the transformed program in the target language emulates the determinate

phase of an Andorra-I execution: the program is executed left-to-right but only determinate goals are reduced, non–determinate goals being suspended. As the reduction of determinate goals makes other goals determinate, the latter are woken (if suspended) and the determinate phase continues. When no more determinate goals are available, `wakeup/2` is reduced and instantiates the first token in the chain–list, thus awakening the leftmost suspended (non–determinate) goal.

Example 2.1 Consider the following Andorra-I program:

```
p:- p1,p2.  q(X).  p1(X).  p2.
p.          q(Y).  p1(Y).
```

Following the transformation mentioned above, we will obtain the transformed program:

```
p(L,L1):- L=[S|L0],det_susp(nonvar(S),p_susp(L0,L1)).
q(X,L,L1):- L=[S|L0],det_susp(nonvar(S),q_susp(X,L0,L1)).
p1(X,L,L1):- L=[S|L0],det_susp(nonvar(S),p1_susp(X,L0,L1)).
p2(L,L1):- L=[S|L0],det_susp((true;nonvar(S)),p2_susp(L0,L1)).

p_susp(L1,L3):- p1(L1,L2),p2(L2,L3). q_susp(X,L,L). p1_susp(X,L,L). p2_susp(L,L).
p_susp(L,L).                         q_susp(Y,L,L). p1_susp(Y,L,L).
```

Consider now the query `:- p,q(X)`. The transformed query will be:

```
:- p(L1,L2),q(X,L2,L3),wakeup(L1,L3).
```

The following trace represents the computation states in the execution of the transformed program. Note that some steps are summarized and the current store is omitted and already applied to the resolvent.

```
⟨p(L1,L2) : q(X,L2,L3) : wakeup(L1,L3),nil⟩
         L1=[Sp|L11]
⟨q(X,L2,L3) : wakeup([Sp|L11],L3),p_susp(L11,L2)⟩
         L2=[Sq|L21]
⟨wakeup([Sp|L11],L3),q_susp(X,L21,L3) : p_susp(L11,[Sq|L21])⟩
         Sp=up
⟨p_susp(L11,[Sq|L21]),wakeup(L11,L3),q_susp(X,L21,L3)⟩
⟨p1(A,L11,L12),p2(L12,[Sq|L21]),wakeup(L11,L3),q_susp(X,L21,L3)⟩
         L11=[Sp1|L111]
⟨p2(L12,[Sq|L21]),wakeup([Sp1|L11],L3),p1_susp(A,L111,L12) : q_susp(X,L21,L3)⟩
         L12 = [Sq|L21]
⟨wakeup([Sp1|L111],L3),p1_susp(A,L111,[Sq|L21]) : q_susp(X,L21,L3)⟩
         Sp1=up
⟨p1_susp(A,L111,[Sq|L21]) : wakeup(L111,L3),q_susp(X,L21,L3)⟩
         L111 = [Sq|L21]
⟨wakeup([Sq|L21],L3),q_susp(X,L21,L3)⟩
         Sq=up
⟨q_susp(X,L21,L3) : wakeup(L21,L3),nil⟩
         L21 = L3
```

$\langle$`wakeup(L3, L3), nil`$\rangle$
$\langle$`nil, nil`$\rangle$

Clearly, several possible optimizations can be applied to this rather general transformation in order to improve execution of the transformed program. But before getting into them, let us first discuss the determinacy conditions (*det_cond* above) and the suspension primitives (for implementing *det_susp*/2 above) in the context of our target language.

3 Determinacy Conditions and Suspension Declarations

The concept of determinacy (of goals, literals, or predicates) has been defined in quite a number of ways in the logic programming community. In general, it is associated to the fact that, for a given goal to a predicate, only one clause will succeed. Informally, a predicate is said to be determinate if all goals complying to the intended use of the predicate are. In a particular program, we will say that a literal is determinate if all goals instantiating that literal are; a predicate being determinate, if all literals in the program are.

Determinacy analysis can be generalized by defining it as the proof of the mutual exclusion of some necessary conditions for the success of the clauses of the predicate definition. This proof can be done based on the environment of a goal, the collecting semantics of a program literal, or that of the predicate itself. This will yield each previous definition of determinacy. Furthermore, the necessary conditions can be safely enforced with several degrees of accuracy, thus giving rise to a whole new range of possibilities for defining determinacy. These conditions are usually expressed in terms of the state of instantiation of the clause heads, and therefore easy to extract from the program text itself. On the other hand, they can be as elaborated as being the result of a global program analysis [26]. In between the former and the latter, one can find a wide range of possibilities. One such kind of conditions are those used in the Andorra-I compiler, which we will also use: *flat determinacy* [6, 29], i.e., that which can be recognized by means of head unification and a simple analysis of built–ins. We will use this exact notion of determinacy for two very practical reasons: on the one hand, we would like to be able to compare our results to those of Andorra-I. On the other hand, we would like to reuse the determinacy detection phase of the Andorra-I compiler in our implementation.

Another relevant issue in the context of determinacy detection is the nature and complexity of the determinacy conditions, which can vary from very simple conditions such as `true` or `false`, to complex conjunctions and/or disjunctions of tests regarding the instantiation states, the type, or the unifiability of some variables. Although our kernel language does include a rich set of such primitives, we find it interesting in this work to restrict that set to those primitives that are available on most general purpose Prolog/CLP systems. Therefore, we will default to, and explain how to substitute the *det_susp* function introduced in the previous section by the usual suspension primitives or user–predicates defining the expected relation.

SICStus Prolog, for example, provides coroutining facilities by means of `block` declarations and `when` meta–calls, among others [3]. The block declaration takes the form:

`:- block` *Spec*`, ...,` *Spec*`.`

where each *Spec* is a mode spec of the goals for the predicate, and each one specifies a condition for

blocking goals of the predicate referred to by it. When a goal for the predicate is to be executed, the mode specs are interpreted as conditions for blocking the goal, and if at least one condition evaluates to *true*, the goal is blocked. A block condition evaluates to *true* iff all arguments specified as `-` are uninstantiated, in which case the goal is blocked until at least one of those variables is instantiated. The `when` meta–call has the general form:

`when(`*Condition*`,`*Goal*`)`

and blocks *Goal* until *Condition* is true, where *Condition* is a Prolog goal given by the following restricted syntax:

$Condition ::= \texttt{nonvar}(\texttt{X})$
$Condition ::= \texttt{ground}(\texttt{X})$
$Condition ::= ?=(\texttt{X},\texttt{Y})$
$Condition ::= Condition,Condition$
$Condition ::= Condition;Condition$

Clearly, using this kind of primitives restricts the kind of suspensions that can be directly expressed. Whenever the determinacy condition *det_cond* only contains `nonvar` tests over the predicate arguments, then a `block` like suspension declaration can be used. It will only be necessary to put *det_cond* in conjunctive normal form and define one mode spec for each conjunct, where each of these will have the corresponding argument replaced by a `-` flag. If this is not possible, but the condition fits into the above syntax, then `when`–like literals can be used. Again, the condition can be put in either conjunctive or disjunctive normal form, each item in this expression replaced by the corresponding checks, and an expression built from the latter. If neither of them can be used, user–defined predicates should be used, which will be the result of compiling the decision graph which corresponds to the condition into Prolog itself.

4 Handling Non-Pure Features

In this section we will discuss the transformation of built-ins. There are basically three kinds of built-ins. The first type is formed by those built-ins which can be considered as normal user goals whose low-level implementation is only due to efficiency reasons. In this case, there are only two differences between the general transformation defined in previous section and that applied to these built-ins[1]. Firstly, the determinacy condition will be obtained from an internal database rather than from the examination of the definition of the predicate. Secondly, the predicate provided as the second argument of *det_susp*/2 will be the original built-in.

Example 4.1 One possible transformation for the >/2 built-in could be the following:

```
>(X,Y,L,L2) :-
      L=[S|L1],
      when((ground(X/Y);nonvar(S)),X>Y).
```

[1] There is no point in including such transformed built-ins in each transformed program. Thus, they will form part of a special module containing also the definition of the `wakeup/2` predicate and some other related predicates.

The second class is formed by those built-ins whose early evaluation may affect the correctness of the execution. For example, side-effect built-ins such as `write/1`, meta-predicates such as `var/1`, or pruning operators such as cuts. In the Andorra-I system this problem is easily solved by just delaying such built-ins until they become leftmost [6]. In our approach, this simply corresponds to creating a *false* determinacy condition.

The third class is formed by built-ins which can be affected by the early execution of goals which are dependent from it. In particular, whenever:

- an early failure prevents the execution of a side-effect, or
- an early binding affect cuts, meta-predicates and side-effects that assume their arguments to be unbound.

Many different approaches can be taken in order to eliminate this problem [6]. We will follow the same approach taken by the Andorra-I system, namely to detect these *sensitive* built-ins and prevent any execution of goals to the right of one such built-in until both all goals to its left and the built-in itself have been completely executed. In order to do this we will make use of "intermediate" wakeup goals. Since, as we will see, the method is simple and intuitive we will avoid its formal definition which seems to require too complex machinery.

The idea is to add a wakeup goal just after the transformed sensitive built-in, thus forcing all goals to the left to be executed before continuing with those to its right. This implies adding (at least) an extra argument to the predicates in order to provide the wakeup goal with the appropriate "initial" pointer. Note that for those built-ins which belong both to the second and third classes, allocating the wakeup goal just *before* the *original* (non transformed) built-in, would both prevent any execution of goals to its right and delay the built-in until it becomes leftmost, thus solving both problems.

Example 4.2 Consider the following piece of a program:

```
p(X):- q(X), r(X),write(X),s(X).
```

An early execution of `write(X)` might affect the correctness of the execution. Thus we will delay it until leftmost. Also, it is a "sensitive" built-in and therefore we must disallow early execution of the goals to the right. The predicate can be transformed as follows:

```
p(X,L,L1,In):-
        L=[S|L0],det_susp(nonvar(S),p_susp(X,L0,L1,In)).

p_susp(X,L1,L4,In):-
        q(X,L1,L2,In), r(X,L2,L3,In),wakeup(In,L3),write(X),
        s(X,L3,L4,L3).
```

Thus the transformation is just changed in that we have to add an extra argument, which will be the first argument of the intermediate `wakeup` goal introduced just before the sensitive built-in. Then, when all goals in the list delimited by `In` and `L3` have been already executed, the second argument of the `wakeup` goal (i.e., `L3`) will become the new left pointer.

Note that with this transformation, wakeup goals outside the derivation tree of the `p(X,L1,L4,In)` will also have `In` as the first argument, even when all variables from `In` to the place initially marked by `L3` are already instantiated to `up`. Therefore, we will be generating useless work. This problem can be solved by adding another extra argument which returns the new left pointer.

Example 4.3 Consider again the predicate of Example 4.2. A less naive transformation would yield the following definitions:

```
p(X,L,L1,In,Out):-
        L=[S|L0],det_susp(nonvar(S),p_susp(X,L0,L1,In,Out)).

p_susp(X,L1,L4,I,O):-
        q(X,L1,L2,I,O1), r(X,L2,L3,O1,O2),wakeup(O2,L3),write(X),
        s(X,L3,L4,L3,O).
```

5 Mixing Prolog and Andorra-I Code

It can be argued that while the Basic Andorra Principle is certainly interesting for its pruning capabilities, in some cases a simple Prolog execution may be desirable. This can be the case for programs known to be deterministic, or for which a fixed ordering of choice points is known to be best, since then the determinacy checking overhead can be avoided (as well as the associated compilation time). One interesting possibility that the transformational approach brings is to mix execution in "Andorra mode" with normal (in this case, Prolog) execution. It is quite easy to call from straight Prolog code to "Andorra transformed" code and the other way around. We will assume that the source is marked in some way to distinguish those predicates that should be compiled as normal Prolog predicates from those that are to be compiled to support the Andorra model. The transformation is then done only on those predicates (files, modules,...) marked as meant to run under the Andorra model. Calls from Prolog to Andorra goals are done in the same way as shown previously for queries: a call to `wakeup/2` is introduced after the goal, so that the whole Andorra computation is completed before continuing the Prolog execution (if this is what is desired – we assume the intended operational behavior is to isolate both executions).[2] Calls from Andorra-I to Prolog are *preceded* by a call `wakeup/2`, and will be then executed normally, outside the context of any Andorra goals (again, if this is what is desired). An interesting alternative, from the point of view of marking Andorra and Prolog execution parts would also be to simply mark certain calls as Andorra calls (by, for example, wrapping them in a `bam/1` goal). The compiler would then simply generate special, transformed versions of all the predicates called by that goals and its descendents (in addition to the normal ones). This avoids having to mark program parts instead.

6 Optimizing the Transformation

Several sources of possible overheads arise in the above transformation. First, the amount of code generated: for each procedure definition, the (generic) transformation yields up to two procedures. Second, the addition of some arguments — at least the pointers to the chain–list of tokens, which

[2] Note that this has some resemblance to a deep guard, as used in AKL [21].

may happen to be unnecessary. Finally, the need for detecting the conditions for determinism and the leftmost goal, and the related suspension on these conditions. In this section we discuss some optimizations that can be performed regarding these three sources, based on information available prior to performing the transformation, and define an improved transformation.

6.1 Simple enhancements

If a goal is determinate at the time it is first considered for execution, it should not be suspended to be immediately woken. Therefore, a translated program which checks the determinacy condition *det_cond*($\bar{X}$) first before blindly suspending with *det_susp*/2 can be more efficient at execution time:

```
p(X̄,L,L1):-
        goal(det_cond(X̄)) -> L = L1, p_det(X̄,L0,L1)
        ; L = [S|L0], det_susp((det_cond(X̄);nonvar(S)),p_susp(X̄,L0,L1)).
```

In general, *det_cond*($\bar{X}$) may not be directly executable in the target language, and thus it must be mapped into suitable goals by *goal*(*det_cond*($\bar{X}$)). If this goal succeeds, a specialized version of `p_susp/n+2` which does not need to suspend, `p_det/n+2`, will be called. Because this predicate will only be called when goals are known to be determinate, it is possible to avoid the creation of choice–points when reducing them with its clauses by appropriately adding cuts. This would closely simulate the commitment to certain clauses introduced by Andorra-I once a reduction is known to be determinate.

6.2 Determinacy condition is true

Analyses can be done which guarantee that a determinacy condition will always succeed, will always fail, or will partly succeed. One such analysis is an abstract interpretation of the original program based on the Prolog semantics. Note that the results of this analysis are still correct for our purposes as long as they approximate a downwards closed property such as groundness, since the differences in our execution model w.r.t. that of Prolog amount to the Andorra model executing goals *ahead* of its turn — in the sense of Prolog's left–to–right execution. In other words, since the determinacy conditions rely on the state of instantiation of variables, and logic languages are monotonic, executing goals ahead of time can only result in further instantiation of the variables.[3] An alternative analysis can be based on the Andorra semantics itself. Furthermore, even a simple local analysis of the program can be performed. We regard such analyses as "a priori" analysis w.r.t. the transformation, since they are performed over the original program.

Predicates whose determinacy can be inferred from examining the head of the clauses and/or simple exclusive built-ins appearing at the beginning of the body, can be easily recognized with such a Prolog semantics–based analysis. The conditions for determinacy of a predicate can be expressed as checks on the degree of instantiation of certain argument variables. A mode/type analysis can then guarantee that that degree of instantiation is always reached at the time of executing a given goal. Let SC be the set of clauses in the definition of a predicate. Let us assume that we associate with each clause $C_i \in SC$ the subset M_i of the set of facts which define the meaning of such clause, the analysis have been able to infer. The condition to be checked is that for a given abstract

[3]This has also been identified in the context of other computational models — see [13].

constraint λ, and for every constraint c approximated by λ, it holds that for every $f_i \in M_i$ and $f_j \in M_j$, $i \neq j$, the constraint $f_i \wedge f_j \wedge c$ is inconsistent.

A simple case of determinate goals, for which nothing more than inspecting the program text is needed, is that of goals of a predicate defined by a unique clause. In this case, a straightforward compile–time optimization can be achieved by a naive one–step unfolding of such goals. Since this one could also be applied to Andorra programs themselves, we do not take it into account, but only (possibly) until after our transformation is done.

When the determinacy condition is reduced to true, it is clear that the general transformation based on an if-then-else defined previously, can be reduced to its "then" part. There is no need to introduce a leftmost–token (i.e., no need to attach any variable), and no need of any suspension. Therefore the extra clause the transformation will add amounts to a simple renaming, which in fact can be performed at transformation–time. Alternatively, once more, it will be achieved by a partial evaluation with a simple one–step unfolding.

6.3 Determinacy condition is false

In this case, a Prolog semantics–based analysis is not safe. Not only that the analysis can not guarantee that if an abstract substitution implies that a variable is not sufficiently instantiated it meant that it will never be at execution time, but also that such an analysis will not take into account that running goals ahead of time can make other goals become determinate.

Nonetheless, from the definitions of certain predicates it can be easily detected that it is not possible that the clauses are exclusive. The condition to be checked for this is that, for the set M_i of facts which are true for each clause C_i of the predicate, it holds that for every $f_i \in M_i$ and $f_j \in M_j$, $i \neq j$, and for every constraint c, the constraint $f_i \wedge f_j \wedge c$ is consistent.

When the determinacy condition is reduced to false, it is clear that the general transformation based on an if-then-else defined previously, can be reduce to its "else" part. The only thing needed is introducing a leftmost–token (i.e., to attach a particular variable), so that goals always suspend until leftmost. In this case the original definition of predicate `p` should not only have to be renamed to `p_susp`, but additionally an extra argument should be added on which to suspend.

6.4 Unchaining calls to predicates

This optimization is based on the observation that certain clauses do not need the extra arguments for the difference list of tokens to be passed. Such clauses include facts, and others which only have literals in their body which, in turn, do not need such arguments, either. A clause is, therefore, said to be *unchained* if all literals in its body are either: constraints (or unification equations), "always–executable" built-ins (such as `true`), or goals for an unchained predicate. A predicate is said to be unchained if all clauses in its definition are unchained and the predicate itself is determinate.

Note that an unchained clause does not need to have the extra arguments because, as soon as it is reduced, the left–to–right execution of its body correspond to the Andorra model, due to the nature of the literals in it. Also, since the previous definitions are recursive, one could think that an analysis for unchain–ness should have to incorporate a fix-point computation. Nonetheless, this is in fact not needed. Let all literals of a clause be unchained, except for a recursive call; the clause

will be unchained only if that call does not need a token to be passed, and this is so only if it is determinate. Therefore, a simpler algorithm in this style, which will allow marking clauses and predicates as unchained, can be defined as follows:

$$
\begin{aligned}
&unchained(\mathtt{p/n}) \leftarrow && constraint(\mathtt{p/n})\\
&unchained(\mathtt{p/n}) \leftarrow && ready_builtin(\mathtt{p/n})\\
&unchained(\mathtt{p/n}) \leftarrow && determinate(\mathtt{p/n}) \wedge \forall C \in defn(\mathtt{p/n})\ unchained(C)\\
\\
&unchained(C) \leftarrow && body(C) = \emptyset\\
&unchained(C) \leftarrow && \forall \mathtt{p/n} \in body(C)\\
& && (\neg recursive(\mathtt{p/n}) \rightarrow\ unchained(\mathtt{p/n}))\wedge\\
& && (recursive(\mathtt{p/n}) \rightarrow\ determinate(\mathtt{p/n}))
\end{aligned}
$$

where $constraint$(p/n) holds if p/n is a constraint symbol, $ready_builtin$(p/n) if it is an "always–executable" built-in, $determinate$(p/n) if predicate p/n is known to be determinate, $recursive$(p/n) if it is recursive, $defn$(p/n) gives the set of clauses defining it, and $body(C)$ gives the list of predicates of the body of a clause C.

Given that some predicates and clauses are marked as unchained, the definition of p_susp/n+2 presented in Section 2 can be modified to take this into account:

- If $unchained$(p/n), then we can avoid the two extra arguments, the definition of p_susp/n being the result of renaming the functor p by the functor p_susp.
- Otherwise, for every clause C in the set of clauses SC defining p/n:
  - If $C \equiv$ p($\bar{\mathtt{X}}$)., it is transformed into p_susp($\bar{\mathtt{X}}$,L,L).
  - If $C \equiv$p($\bar{\mathtt{X}}$):- q1($\bar{\mathtt{Y1}}$) $\cdots$ qn($\bar{\mathtt{Yn}}$)., with $n > 0$, and $\exists i \in [1, n]$ $unchained$(qi/ni), C is transformed into the clause p_susp($\bar{\mathtt{X}}$,L1,Ln):- Q1, $\cdots$,Qm. where $\mathtt{m} = \mathtt{n} - 1$, and $\mathtt{Q1} = \begin{cases} \mathtt{qi(\bar{Yi}),\ Li=Lj} & if\ unchained(\mathtt{qi/ni}) \\ \mathtt{qi(\bar{Yi},Li,Lj)} & otherwise \end{cases}$ where $\mathtt{j} = \mathtt{i} + 1$

Obviously the unification equations can be solved during the transformation, substituting one variable for the other. In the following we will denote by $chain(SC)$ the function which transforms the set of clauses SC following the above proposed method.

6.5 An optimized algorithm

Note that the optimizations presented can be defined in terms of program specialization and code reduction, based on concepts of abstract executability [12]. Having such a specializer, together with a simple partial evaluator, the transformation proposed can default to the most general one, and be done blindly. However, because doing the whole process in one single step can be more efficient, and also because some unfoldings can be done which relate to predicates affected by delay declarations (something that a partial evaluator will sure not consider), we present a generic algorithm performing the translation from the original program and achieving all the optimizations.

Let SC be the set of clauses which define predicate p/n, with most general goal p($\bar{\mathtt{X}}$), and $C(\bar{\mathtt{X}})$ be the determinacy condition w.r.t. p($\bar{\mathtt{X}}$), already simplified making use of the information available. The transformation will substitute SC by a new set of clauses SC' as follows:

- If $C(\bar{\mathtt{X}}) = true$ and $unchained(\mathtt{p/n})$ holds, the predicate will never suspend and no goal will be suspended during its execution. Therefore, we need neither suspension conditions, nor attached variables, nor chain pointers. Thus, $SC' = SC$.

- If $C(\bar{\mathtt{X}}) = true$ but $unchained(\mathtt{p/n})$ does not hold, the predicate will never suspend but goals might suspend during its execution. Thus we might need chain pointers. Therefore, $SC' = chain(SC, \mathtt{p})$.

- If $C(\bar{\mathtt{X}}) = false$ and $unchained(\mathtt{p/n})$ holds, the predicate will always be initially suspended but, once it has been woken, no goal will be suspended during its execution. Thus, we will just need to attach a variable and place a condition on the instantiation state of such variable. Therefore SC' is equal to SC plus the following clauses:

```
p(X̄,[S|L],L):- p_susp(X̄,S).
:- block p_susp(?̄,-).
p_susp(X̄,S):- p(X̄).
```

- If $C(\bar{\mathtt{X}}) = false$ and $unchained(\mathtt{p/n})$ does not holds, we might also have to add chain pointers. Thus, will just need attached variables. Therefore SC' is equal to $chain(SC, \mathtt{p_work})$ plus the following clauses:

```
p(X̄,[S|L0],L1):- p_susp(X̄,S,L0,L1).
:- block p_susp(?̄,-,?,?).
p_susp(X̄,S,L0,L1):- p_work(X̄,L0,L1).
```

- Otherwise, SC' is formed by the following clauses:

$$
\begin{array}{ll}
\mathtt{p(\bar{X},L,L1)\text{:-}} & goal(C(\bar{\mathtt{X}}))\ \mathtt{\text{->}}\ work_goal(\mathtt{p(\bar{X})},\mathtt{L},\mathtt{L1}) \\
 & \mathtt{L=[S|L0]},\ det_susp(C(\bar{\mathtt{X}}),\mathtt{p(\bar{X})},\mathtt{S},\mathtt{L0},\mathtt{L1}). \\
\mathtt{work_def(SC,p(\bar{X}))} &
\end{array}
$$

where $work_goal$, $work_def$, and det_susp are defined as follows:

$$
work_def(SC, \mathtt{p(\bar{X})}) = \begin{cases} SC & if\ unchained(\mathtt{p/n}) \\ chain(rename(SC, \mathtt{p_work})) & otherwise \end{cases}
$$

$$
work_goal(\mathtt{p(\bar{X})}, \mathtt{L}, \mathtt{L1}) = \begin{cases} \mathtt{p(\bar{X}),\ L=L1} & if\ unchained(\mathtt{p/n}) \\ \mathtt{p_work(\bar{X},L,L1)} & otherwise \end{cases}
$$

$$
det_susp(C(\bar{\mathtt{X}}), \mathtt{p(\bar{X})}, \mathtt{S}, \mathtt{L0}, \mathtt{L1}) =
$$

$$
\begin{cases}
if\ block(C(\bar{\mathtt{X}})) & \begin{cases} \mathtt{p_susp(\bar{X},S,L0,L1).} \\ \mathtt{:\text{-}\ block\ p_susp(}block(C(\bar{\mathtt{X}}))\mathtt{,\text{-},?,?).} \\ \mathtt{p_susp(\bar{X},S,L0,L1)\text{:-}}\ work_goal(\mathtt{p(\bar{X})},\mathtt{L0},\mathtt{L1}). \end{cases} \\
if\ when(C(\bar{\mathtt{X}})) & \mathtt{when(\ (}\ when(C(\bar{\mathtt{X}}))\ \mathtt{;\ nonvar(S)\ ),}\ work_goal(\mathtt{p(\bar{X})},\mathtt{L0},\mathtt{L1})\ \mathtt{)}
\end{cases}
$$

where $chain(SC, \mathtt{f})$ is identical to the function $chain(SC)$ defined before but using the functor `f` instead of `p_susp`, $goal(C(\bar{\mathtt{X}}))$ gives the Prolog goal corresponding to a determinacy condition $C(\bar{\mathtt{X}})$, $block(C(\bar{\mathtt{X}}))$ gives the sequence of block annotations corresponding to $C(\bar{\mathtt{X}})$ or fails if it is not possible to do so, and $when(C(\bar{\mathtt{X}}))$ does the same for when annotations. These functions have been informally defined in Section 3.

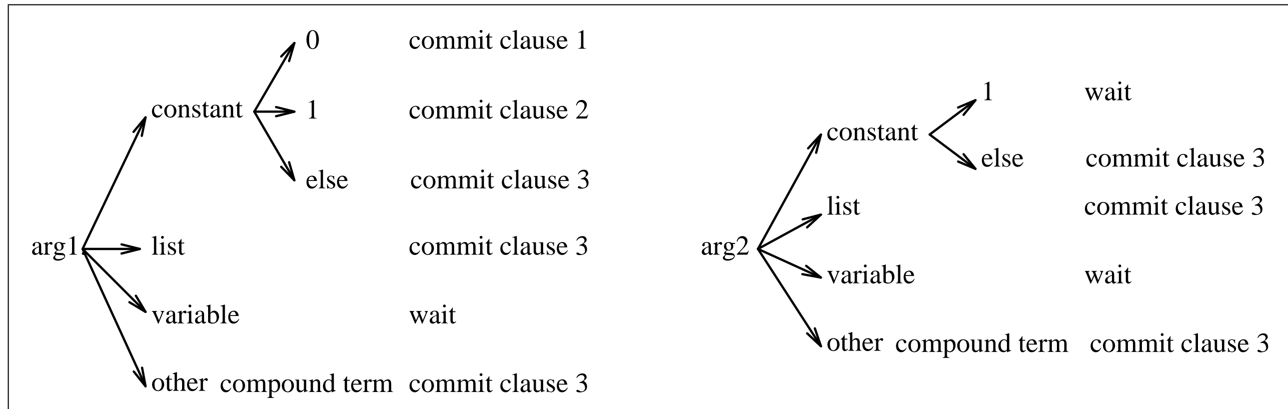


Figure 2: Andorra-I Decision Graph.

7 An Example

In this section we will illustrate the transformation and optimization procedure by means of a detailed simple example, the well known program for computing the Fibonacci series.

```
fib(0,1).
fib(1,1).
fib(A,B) :- A>1, C is A-1, D is A-2, fib(C,E), fib(D,F), B is F+E.
```

The decision graph created by the Andorra-I preprocessor for this program is the one shown in Figure 2. Given that graph, it easy to conclude that `fib(X,Y)` will be determinate as soon as (a) `X` become non–variable or (b) `Y` become a term not unifiable to 1. If only condition (a) had been necessary, a block suspension primitive would had been enough.

```
fib(X,Y,L,L2) :- L = [S|L1], fib_susp(X,Y,S,L,L2).

:- block fib_susp(-,?,-,?,?).

fib_susp(X,Y,S,L1,L2) :- fib_work(X,Y,L1,L2).

fib_work(0,1,L1,L1).
fib_work(1,1,L1,L1).
fib_work(A,B,L1,L2) :- >(A,1,L1,L3), is(C,A-1,L3,L4), is(D,A-2,L4,L5),
                       fib(C,E,L5,L6), fib(D,F,L6,L7), is(B,F+E,L7,L2).
```

However, given the need of a non–unifiability test, even a `when` declaration is not enough. Thus, the transformation needs to perform an explicit checking on the conditions the above graph represents.

```
fib(X,Y,L,L2) :- var(X), !, fib0(X,Y,L,L2).
fib(X,Y,L,L2) :- fib_work(X,Y,L,L2).
```

```
fib0(X,Y,L,L2):- var(Y), !, L=[S|L1], fib_susp12(X,Y,S,L1,L2).
fib0(X,Y,L,L2):- fib1(Y,X,L,L2)

fib1(1,X,L,L2):- !,
        L=[S|L1],
        fib_susp1(X,1,S,L1,L2).
fib1(Y,X,L,L2):-
        fib_work(X,Y,L,L2).

:- block fib_susp1(-,?,-,?,?).

fib_susp1(X,Y,S,L1,L2) :- fib_work(X,Y,L1,L2).

:- block fib_susp12(-,-,-,?,?).

fib_susp12(X,Y,S,L1,L2) :- nonvar(S), !, fib_work(X,Y,L1,L2).
fib_susp12(X,Y,S,L1,L2) :- fib(X,Y,L1,L2).
```

Given a(n) (abstract) query where the first two arguments are known to be ground, an abstract interpretation–based analysis is enough to determine that those two arguments will be ground for all calls to `fib/2`, and thus that it is determinate. Furthermore, it will also determine that the built-ins are directly reducible. Therefore, this rather complex transformation can be avoided, and the original Prolog program could be used instead.

An analysis based on a semantics with dynamic scheduling, such as for example [25], can provide the same information. Furthermore, since the kind of information derived by such analysis does not need to be downwards closed, it can determine that for a given query with first two arguments free and the third ground (which, given the transformed program, is perfectly possible), no goal is ever determinate (i.e. the goals are only awakened on the attached variable `S`). Therefore, suspensions can be further reduced, giving the following program:

```
fib(X,Y,L,L2) :- L=[S|L1], fib_susp12(X,Y,S,L1,L2).

:- block fib_susp12(?,?,-,?,?).

fib_susp12(X,Y,S,L1,L2) :- fib_work(X,Y,L1,L2).
```

8 Performance Figures

In this section we present the results obtained from a preliminary evaluation of the proposed approach. The experiments aim at determining the efficiency of the transformed programs when compared to the Andorra-I programs.

The evaluation has been performed using a preliminary implementation of the transformation procedure. The optimizations performed during such transformation are the following. First, it introduces the necessary code to avoid a goal to be suspended if it is determinate at the moment it is processed, as explained in Section 6.1. Second, it unchanges the predicates following the algorithm provided in Section 6.4. Finally, it takes into account the cases in which the determinacy conditions are `true` or `false`.

Bench	Andorra-I	SICStus	
		Compiled	Interpreted
fib	260	127	643
map	103	101	244
mutest	236	138	531

Table 1: Execution times in milliseconds

In order to be self-contained, the current system includes a simple determinacy condition generator. However, in order to make sure that the same reductions are performed by both systems being compared in our experiments, and because in the future we plan to derive determinacy conditions by using the information provided by the Andorra-I preprocessor, we have modified by hand the transformed program to implement determinacy conditions which accurately simulate those produced by the Andorra-I preprocessor.

Table 8 shows for some Andorra-I benchmark the execution time in Andorra-I and in SICStus, using the transformation. For the latter, two cases are considered: when the program is compiled and when it is interpreted. The implementation technology of the Andorra-I system available to us (which is not the latest, compiler-based version), is somewhere between a compiler and an interpreted system, in the sense that it is an optimized interpreter written directly in C (rather than a slower, source-level meta-interpreter, as in the Prolog interpreter). Although it is clear that no conclusions can be derived from only three benchmarks, we believe that the results are encouraging since the performance of the resulting system is so far comparable to the native Andorra-I implementation.

9 Conclusions

We have reported on a transformation technique which allows supporting the Andorra computational model, essentially emulating the Andorra-I system, via program transformation into a sequential language with delay primitives. We have also proposed several optimizations to the transformation. The system is automatic, comprising a simple program analyzer and a basic transformer to the kernel language. The preliminary results are quite encouraging: performance of the resulting system is comparable to the current Andorra-I implementation.

We do not mean to suggest that the performance of a system implemented using our approach is optimal or that it will achieve in the end better results than a highly optimized, native the native Andorra-I implementation, but rather that the technique is practical and allows the support of the Basic Andorra Model on a generic system with reasonable performance. This is specially useful in view of the proposed methods for combining traditional Prolog (or CLP) code and Andorra code.

We plan to further optimize and benchmark the system. We would also like to explore the impact of certain optimizations which are applicable to the transformed programs, rather than to the source programs, and which deal mainly, but not only, with the avoidance of suspensions. Since the transformed program is simply a Prolog (or CLP) program with delay, standard analyses for these kinds of programs can be used [25]. We also plan on performing other optimizations such as further simplifying and optimizing the determinacy conditions and eliminating dead code. Finally,

we are also planning on testing performance on parallel systems. Note that or-parallelism comes for free by simply running the transformed program on a system like Muse [1] or Aurora [24]. We also plan on implementing and-parallelism (both determinate–dependent and also independent) by using the recently proposed notions of independence in systems with delay [5], the associated compilation technology, and the parallel implementation of the *Ciao* system.

References

1. K.A.M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*. MIT Press, October 1990.
2. M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *Fourth International Conference on Logic Programming*, pages 40–58. University of Melbourne, MIT Press, May 1987.
3. M. Carlsson. *Sicstus Prolog User's Manual*. Po Box 1263, S-16313 Spanga, Sweden, February 1988.
4. Jim Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In *1990 North American Conference on Logic Programming*. MIT Press, 1990.
5. María José García de la Banda García. *Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming*. PhD thesis, Universidad Politécnica de Madrid (UPM), July 1994.
6. Vítor Manuel de Morais Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I*. PhD thesis, University of Bristol, August 1993.
7. S. K. Debray. Implementing logic programming systems: The quiche-eating approach. In *ICLP '93 Workshop on Practical Implementations and Systems Experience in Logic Programming*, Budapest, Hungary, June 1993.
8. S. K. Debray. QD-Janus : A Sequential Implementation of Janus in Prolog. *Software—Practice and Experience*, 23(12):1337–1360, December 1993.
9. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.
10. I. Foster and S. Taylor. *Strand* : A practical parallel programming tool. In *1989 North American Conference on Logic Programming*, pages 497–512. MIT Press, October 1989.
11. P. López García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In *Proc. of PASCO'94*. World Scientific Publishing Company, September 1994.
12. F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, pages 323–335. Springer-Verlag, 1991.
13. G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: Integrating Dependent and-, Independent and-, and Or-parallelism. In *1991 International Logic Programming Symposium*, pages 152–166. MIT Press, October 1991.
14. M. Hermenegildo, D. Cabeza, and M. Carro. On The Uses of Attributed Variables in Parallel and Concurrent Logic Programming Systems. In *Proc. of the 1994 COMPULOG-NET Workshop Parallelism and Implementation Technologies*. U. of Madrid, September 1994. Also provided as attachment UPM-1 of deliverable D4.3/2.

15. M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

16. M. Hermenegildo and the CLIP group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel Concurrent Constraint System. In *Proc. of the 1994 Workshop on the Principles and Practice of Constraint Programming*. U. of Washington, May 1994.

17. C. Holzbaur. *Specification of Constraint Based Inference Mechanisms through Extended Unification*. PhD thesis, University of Vienna, 1990.

18. C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *1992 International Symposium on Programming Language Implementation and Logic Programming*, pages 260–268. LNCS631, Springer Verlag, August 1992.

19. A. Houri and E. Shapiro. A sequential abstract machine for flat concurrent prolog. Technical Report CS86-20, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel, July 1986.

20. Serge Le Huitouze. A New Data Structure for Implementing Extensions to Prolog. In P. Deransart and J. Małuszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 136–150. Springer, August 1990.

21. S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. Technical Report PEPMA Project, SICS, Box 1263, S-164 28 KISTA, Sweden, November 1990. Forthcoming.

22. Andy King and Paul Soper. Schedule Analysis of Concurrent Logic Programs. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 478–492, Washington, USA, 1992. The MIT Press.

23. S. Kliger. *Compiling Concurrent Logic Programming Languages*. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, October 1992.

24. E. Lusk et. al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.

25. K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *Proceedings of the 20th. Annual ACM Conf. on Principles of Programming Languages*. ACM, January 1994.

26. C.R. Ramakrishnan S. Dawson and I.V. Ramakrishnan. Extracting Determinacy in Logic Programs. In *1993 International Conference on Logic Programming*, pages 424–438. MIT Press, June 1993.

27. V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, April 1990.

28. V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 825–839. MIT Press, June 1991.

29. V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

30. J. Thom and J. Zobel. *NU-Prolog Reference Manual*. Dept. of Computer Science, U. of Melbourne, May 1987.

31. E. Tick. Compile-Time Granularity Analysis of Parallel Logic Programming Languages. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

32. E. Tick and C. Bannerjee. Performance evaluation of monaco compiler and runtime kernel. In *1993 International Conference on Logic Programming*, pages 757–773. MIT Press, June 1993.

33. K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.