Improving Deterministic Computations in Lazy Functional Logic Languages

Rafael Caballero and Francisco J. López-Fraguas *

Dpto. de Sistemas Informáticos y Programación, Universidad Complutense de Madrid
e-mail: {rafa,fraguas}@sip.ucm.es

Abstract. The detection of deterministic computations at run-time can be used to introduce *dynamic cuts* pruning the search space and thus increasing the efficiency of functional logic systems. This idea was introduced in an early work of R. Loogen and S. Winkler. However the proposal of these authors cannot be used in current implementations because it did not consider non-deterministic functions and was not oriented to the demand driven strategy. Our work adapts and extends the technique, showing both how to deal with non-deterministic computations, and how definitional trees can be employed to locate the places where the cuts will be introduced. An implementation based on a Prolog-translation is proposed, making the technique easy to implement in current systems generating Prolog code. Some experiments showing the effectiveness of the cut are presented.

1 Introduction

Efficiency has been one of the major drawbacks associated with declarative languages. The problem becomes particularly severe in the case of Logic Programming (LP for short) and Functional Logic Programming (FLP for short), where the introduction of non-deterministic computations often generates large search spaces with their associated overheads both in terms of time and space.

The idea of detecting deterministic computations is not new in declarative programming. In the case of the LP language Prolog [10], a non-declarative mechanism, the so-called *cut*, has been introduced. Programs using cuts become much more efficient, but at the price of becoming non-declarative. Other works [9, 8] propose declarative alternatives avoiding unnecessary re-evaluations automatically.

Rita Loogen and Stephan Winkler presented in [15] a technique for the run-time detection of deterministic computations that can be used to safely prune the search space in functional logic programs. This technique is known as *dynamic cut*. The framework of these authors was the FLP programming language BABEL [19]. Unfortunately the programs considered in that work did not include non-deterministic functions, which are used extensively in FLP nowadays. Also

* Work partially supported by the Spanish CYCIT (project TIC2002-01167 'MELODIAS').

the implementation (based on a modification of an abstract machine) did not follow the demand driven strategy [4, 14], which has since been adopted by all the current implementations of FLP languages.

Modern functional logic languages like Curry [13] or $\mathcal{TOY}$ [16, 1] use non-deterministic functions as a common programming resource allowing to program in a very concise, yet efficient way, many problems involving search. An early work in this sense is [20], where it is shown that by a suitable use of the combination *non-deterministic functions + lazy evaluation*, large parts of search spaces can be early pruned, thus resulting in a remarkable gain of efficiency.
Despite of these programming techniques, in the real practice of FLP there are nevertheless many occasions where the generated search spaces are larger than necessary. Or even that search can be completely avoided, if not in all uses of a given program, at least in concrete computations which are deterministic from the point of view of the result. It is in these cases when the idea of dynamic cut becomes useful.

Our proposal adapts the original idea of [15] to FLP languages with non-deterministic functions, which introduce some subtle changes in the conditions for the cut. These dynamic cuts can be easily introduced in a Prolog-based translation of FLP programs that uses definitional trees. This makes our technique easily adaptable to the current implementations of FLP languages based on translation into Prolog code. The result of implementing the dynamic cut is more efficient executions in the case of deterministic computations and with no serious overhead in the case of non-deterministic ones, as shown in the runtime table of Section 6.

The aim of this paper is eminently practical and the technique for introducing dynamic cuts is presented in a (hopefully) precise but not formal way.

In the following section, after some preliminaries, we describe at the practical level the operational procedure of FLP by means of a translation into Prolog. Section 3 discusses several examples motivating the introduction of the dynamic cut in the implementation of FLP programs. Section 4 introduces more precisely the key concept of deterministic function, as well as the conditions under which dynamic cut can be safely used. Section 5 discusses an implementation of the technique as a modification of the translation scheme. Section 6 presents a table with the times obtained for the execution, with and without dynamic cut, of some examples. Finally Section 7 presents some conclusions and future work.

2 Operational Procedure of Functional Logic Programming: The Prolog Based Approach

2.1 Preliminaries

All the examples in this paper are written in the concrete syntax of the lazy FLP language $\mathcal{TOY}$ [16, 1] but can be easily adapted to other FLP languages like Curry [13]. In the sequel, we always suppose a given signature $\Sigma = \langle DC, FS \rangle$,

where $DC = \bigcup_{n\in\mathbb{N}} DC^n$ and $FS = \bigcup_{n\in\mathbb{N}} FS^n$ are ranked sets of *data constructors* resp. *defined function symbols*. We also assume the existence of a countable set Var of variables. Variables and constructor symbols serve to build *constructor terms*, while in *expressions* also function symbols can occur.

A $\mathcal{TOY}$ program is composed of data type declarations, type alias, infix operators, function type declarations and defining rules for functions symbols. Each defining rule for a function $f \in FS^n$ has a *left-hand side*, a *right-hand side* and an optional *condition*:

$$(R) \qquad \underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \ = \ \underbrace{r}_{\text{right-hand side}} \ \Leftarrow \underbrace{e_1 == e'_1, \ldots, e_k == e'_k}_{\text{condition}}$$

where e_i, e'_i and r are expressions (that can contain new extra variables) and each t_j is a constructor term[1] with no variable occurring more than once in different t_k, t_l. If the condition is empty then $\Leftarrow$ is omitted, and an equality in the condition of the form $e == true$ is usually abbreviated to simply e.
We consider goals as expressions e and *answers* as pairs (t, σ) where t is a constructor term representing a result obtained by evaluating e, and σ a substitution of terms for variables such that $dom(\sigma) \subseteq vars(e)$. Notice that this notion of goal, suitable for this work, is compatible with usual goals in $\mathcal{TOY}$ which are of the form: $e_1 == e'_1, \ldots, e_k == e'_k$, simply by assuming that an auxiliary function: main $R_1 \ldots R_n$ = true <== $e_1 == e'_1, \ldots, e_k == e'_k$ is introduced, with $\{R_1, \ldots R_n\} = vars(e_1) \cup vars(e'_1) \cup \ldots \cup vars(e_k) \cup vars(e'_k)$, and then evaluating the goal main $R_1 \ldots R_n$. The introduction of main is also helpful since it extends the application of dynamic cuts to goals, converted in this way to the general case of program functions.

Figure 1 shows an example of $\mathcal{TOY}$ program. This simple program can be used to execute simple queries for finding substrings in a given text. A goal of the form matches (single S) Text succeeds if the string S is part of Text, failing otherwise. A goal matches (and S S') Text succeeds whenever both S and S' are part of Text, while matches (or S S') Text indicates that either S or S' (or both) are part of Text. Function matches relies on function part which checks if X is a substring of Y by looking for two existential variables U and V such that Y results of the concatenation of U, X and V. The labels $(R_1) \ldots (R_7)$ are not part of the program. They are included because they will be used to distinguish the program rules below.

Each function is assumed to have a *definitional tree* [2, 14] with nodes *or, case* and *try*. However, in our setting we will not allow 'multiple tries', i.e. *try* nodes with several program rules, replacing them by nodes *or* with multiple *try* children nodes, one for each rule included in the initial multiple *try*. The tree obtained

[1] Actually in $\mathcal{TOY}$ programs the t_j's can be *patterns*, which are more general than constructor terms, since they include the possibility of partial application of function (or constructor) symbols. But this issue, related to the treatment of HO, is of no relevance to this paper.

```
infixr 50 ++
(R1) [] ++ Ys                  = Ys
(R2) [X|Xs] ++ Ys              = [X|Xs ++ Ys]

(R3) part X Y                  = true <==U ++ X ++ V == Y

data query = single [char] | and query query | or query query

(R4) matches (single S) Text  = true <== part S Text
(R5) matches (and S S') Text = true <== matches S Text, matches S' Text
(R6) matches (or S S') Text   = true <== matches S Text
(R7) matches (or S S') Text   = true <== matches S' Text
```

Fig. 1. Simple Queries

by this modification is obviously equivalent and will be more suitable for our purposes.

For instance a definitional tree for the function matches could be:

$$
\begin{array}{ll}
dt(matches) \equiv matches(X, Text) \rightarrow & \\
\quad \textbf{case } X \textbf{ of} & \\
\quad \langle \text{ single} & : \textbf{try } (R_4) \\
\quad \;\; \text{and} & : \textbf{try } (R_5) \\
\quad \;\; \text{or} & : \text{ matches(or S S', Text)} \rightarrow \textbf{or } \langle \textbf{try } (R_6) \\
& \qquad\qquad\qquad\qquad\qquad\quad \mid \textbf{try } (R_7) \rangle \; \rangle
\end{array}
$$

The semantics of our programs has been presented in previous works (see [12] for instance), where a suitable semantic calculus has been presented as a convenient framework for lazy functional logic languages.

However, in this paper we will pay attention to the *operational semantics* of lazy functional logic languages, since our work introduces a change in the operational behavior of programs. We assume that goals are solved by means of an operational mechanism based on needed narrowing with sharing [3, 4, 14], as well as a Prolog-based implementation as described in [14], consisting mainly in a translation of source $\mathcal{TOY}$ programs into object Prolog programs. A main component of the operational mechanism is the computation of *head normal forms* (hnf) for expressions. A head normal form is either a variable or a constructor-rooted expression.

We will take as convenient concrete description of the operational semantics of a Toy program the semantics of its correspondent Prolog program, which in particular implies that depth-first search with chronological backtracking is used. In Section 5 we will discuss the modification of this translation to introduce the

dynamic cuts and argue about its correctness w.r.t. the translation described next.

2.2 The Translation Scheme

The translation scheme from $\mathcal{TOY}$[2] to Prolog, which is fully detailed in [1], is the result of three stages:

• The source $\mathcal{TOY}$ program, which uses higher order syntax, is translated into $\mathcal{TOY}$-like programs written in first order syntax, following the ideas in [21, 11]

• The compiler introduces *suspensions* [7, 14] into first order $\mathcal{TOY}$ programs. The idea of *suspensions* is to replace each subexpression in right-hand sides of rules with the shape of a function call $f(e_1, \ldots, e_n)$ by a Prolog term of the form $susp(f(e_1, \ldots, e_n), R, S)$ (called a suspension) where R and S are initially (i.e, at the time of translation) new Prolog variables. During execution, parameter passing may produce many 'long distance' copies of a given suspension. If at some step of the execution the computation of a head normal form for $f(e_1, \ldots, e_n)$ occurs, the variable R will be bound to the obtained value, and we say that the suspension has been evaluated. Since unification is a global operation in Prolog, all the copies of the suspension will share, through their own copy of R, this computed value. The argument S in a suspension is a flag to indicate if the suspension has been evaluated or not. Initially S is a variable (indicating a non-evaluated suspension), which is set to a concrete value, say `hnf`, once the suspension is evaluated.

• Finally the Prolog clauses which are the final result of the translation are generated, adding suitable code for *strict equality* and *hnf* (to compute head normal forms). To compute a hnf for an unevaluated suspension *susp(f(X1,...,Xn),R,S)*, a call *f(X1,...,Xn,H)* is made to a specific predicate returning in H the desired head normal form. These are exactly the predicates affected by the introduction of dynamic cut.

Next we explain in more detail the third phase (code generation), since this will be the phase affected by the introduction of dynamic cuts.

Given a function f (already in first order syntax and with suspensions replacing the function calls), the Prolog code associated to f will be represented as $prolog(f, dt(f))$, where the auxiliary function $prolog/2$ takes a definitional tree and a function symbol, possibly different from the function of the definitional tree (this is to introduce new auxiliary functions), returning as value a set of Prolog clauses. The Prolog code $prolog(f, dt)$ is obtained by generating code corresponding to the root of the tree, and then descending recursively in the branches. We distinguish cases according to the shape of the root of dt.

Case 1 (the root is a **case** node):

Assume $dt \equiv f(\overline{s}) \rightarrow \mathbf{case}\ X\ \mathbf{of}\ \langle c_1 :\ dt_1 \ldots c_m :\ dt_m \rangle$

[2] For the sake of simplicity, we consider here a simplified version of $\mathcal{TOY}$ not taking into account *disequality constraints* [5, 1]

In this case different branches correspond to incompatible cases in a given position.

$$prolog(g, dt) = \{\ \mathsf{g}(\overline{s},\mathsf{H})\ \mathsf{:-}\ \mathsf{hnf}(\mathsf{X},\mathsf{HX}),\ \mathsf{g'}(\overline{s}\sigma,\mathsf{H}).\}\ \cup \\ prolog(g', dt_1)\ \ldots \cup prolog(g', dt_m)$$

where $\sigma = \mathsf{X}/\mathsf{HX}$ and $\mathsf{g'}$ is a new function symbol.

Case 2 (the root is an **or** node):
Assume $dt \equiv f(\overline{s}) \rightarrow \mathbf{or}\ \langle dt_1 \mid \ldots \mid dt_m \rangle$

$$prolog(g, dt) = \{\ \mathsf{g}(\overline{s},\mathsf{H})\ \mathsf{:-}\ \mathsf{g}_1(\overline{s},\mathsf{H}).\} \cup\ \ldots \cup\ \{\ \mathsf{g}(\overline{s},\mathsf{H})\ \mathsf{:-}\ \mathsf{g}_m(\overline{s},\mathsf{H}).\} \cup \\ prolog(g_1, dt_1) \cup\ \ldots\ \cup prolog(g_m, dt_m)$$

where $g_1, \ldots, g_m$ are new function symbols.

Case 3 (the tree is a leaf **try**):

Assume $dt \equiv \mathbf{try}\ R$, where R is a program rule

$$f(\overline{s})\ =\ e \Leftarrow l_1 == r_1, \ldots, l_n == r_n$$

Then:

$$prolog(g, dt) = \{\ \mathsf{g}(\overline{s},\mathsf{H})\ \mathsf{:-}\ \mathsf{equal}(l_1,r_1),\ \ldots,\ \mathsf{equal}(l_n,\ r_n),\ \mathsf{hnf}(e,\mathsf{H}).\ \}$$

As an example, Figure 2 shows the translation of the function matches. The correspondence between the definitional tree of matches and that code is straightforward and can be easily checked. Actually, the Prolog code generated by $\mathcal{TOY}$ has several optimizations that we do not show here for simplicity. It is worth pointing out that the inclusion of these optimizations is compatible with the dynamic cut.

3 Motivating Examples

In this section we present some motivating examples showing informally the two situations where dynamic cuts can be useful: the first one is associated to *or* nodes in the definitional trees of semantically deterministic functions, while the second one is associated to existential conditions in program rules. These examples as well as several others can be found at:
`http://babel.dacya.ucm.es/rafa/cut`.

Example 1: Parallel and
Figure 3 shows a correct way of defining the *and* connective in FLP programming, known as *parallel and*.
A definitional tree for this function is:

$$dt(\&\&) \equiv X\ \&\&\ Y\ \rightarrow \\ \mathbf{or}\ \langle \mathbf{case}\ X\ \mathbf{of} \\ \langle\ \text{false}\ : \text{false}\ \&\&\ Y\ \rightarrow \mathbf{try}\ R_1 \\ \text{true}\ : \text{true}\ \&\&\ Y\ \rightarrow \mathbf{case}\ Y\ \mathbf{of}\ \langle true : \mathbf{try}\ R_3 \rangle \rangle \\ \mid \mathbf{case}\ Y\ \mathbf{of}\ \langle false : \mathbf{try}\ R_2 \rangle\ \rangle$$

```
matches(X, Text, H) :-
      hnf(X, HX),
      matches1(HX, Text, H).

% the three possibilities of the 'case' branch
matches1(single(S), Text, H) :-
      equal(susp(part(S, Text), R, Flag),true)
      hnf(true,H).

matches1(and(S, S'), Text, H) :-
      equal(susp(matches(S, Text), R1, Flag1),true),
      equal(susp(matches(S', Text), R2, Flag2),true),
      hnf(true,H).

matches1(or(S,S'), Text, H) :-
      matches1'(or(S,S'), Text, H).

matches1(or(S,S'), Text, H) :-
      matches1''(or(S,S'), Text, H).

% the two possibilities of the 'or' branch
matches1'(or(S,S'), Text, H) :-
      equal(susp(matches(S, Text), R, Flag),true),
      hnf(true,H).

matches1''(or(S,S'), Text, H) :-
      equal(susp(matches(S', Text), R, Flag),true),
      hnf(true,H).
```

Fig. 2. Generated Prolog Code

A goal like false && false returns false as expected, but unnecessarily repeats the answer twice:

```
>false && false
false
more solutions? y
false
more solutions? y
no
```

Obviously the computation resulting in the second false was not needed and in this case could have been avoided. The definitional tree shows why: the *or* branch at the top of the tree means that the computations must try both alternatives. In spite of this *or* branch the function will be recognized in our proposal (as well as it was in [15]) as *semantically deterministic*, which means that if the first

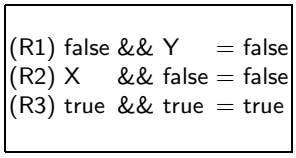

Fig. 3. The parallel and

alternative of the *or* succeeds the other branch either fails or provides a repeated result. The dynamic cut will skip the second branch (under certain conditions) if the first branch is successful, thus avoiding the waste of space and time required by the second computation.

However, as noticed in [15], the cut cannot be performed in all computations. For example, a goal like X && Y will return three different answers:

```
X==false ⇒ false
more solutions? y
Y==false ⇒ false
more solutions? y
X==true, Y==true ⇒ true
more solutions? y
no
```

That is, the result is true if both X and Y are true and false if either X==false or Y==false. Here the second branch of the *or* node contributes to the answer by instantiating variable Y and hence should not be avoided. Therefore the dynamic cut must not be performed if the first successful computation binds any variable; in this case the second computation can eventually instantiate the variables in a different way, thus providing a different answer.

The situation complicates in a setting with non-deterministic functions. Consider for instance the function definition:

```
maybe = true
maybe = false
```

and the goal true && maybe. In this case no variable is bound during the first computation (the goal is ground) but the second computation is still necessary due to the second value returned by maybe

```
true
more solutions? y
false
more solutions? y
no
```

Thus we shall extend the conditions for performing dynamic cuts, requiring not only that no variable has been bound but also that no non-deterministic function has been evaluated. As we will see, this introduces no serious overhead in the implementation.

Example 2
The second example is based on the program of Figure 1, and shows the second type of dynamic cut. The function part is again semantically deterministic but will produce as many repeated results true as occurrences of X can be found. The dynamic cut can be introduced after the conditional part of the rule, since its re-evaluation cannot contribute to new results. Notice that in this case the binding of U and V should not prevent the cut because they cannot contribute to the final substitution σ. In contrast a binding of X or Y will take part of the answer, avoiding the cut.

The effectiveness of the dynamic cut in part is still more noticeable because its effect over the function matches. Assume that there is no dynamic cut, and that we try a goal like

```
matches  (or (and (single "cut") (single "love")) (single "dynamic"))
         "Efficiency has been one of the ..."
```

where the text used as second argument is actually the whole introduction of this paper. Since the query is an *or* query, matches first tries the first alternative, (and (single "cut") (single "love")). Although "cut" is readily found there is no "love" in our introduction and part fails in a first attempt, after examining the whole text. Because of backtracking, a new occurrence of "cut" is sought and found (there are many occurrences of "cut" in the text), and then again part looks unsuccessfully for "love". The process repeats the examination of all the text looking for any occurrence of "love" as many times as occurrences of "cut" exist, therefore spending a huge amount of time before failing. Then the second alternative of the *or* query succeeds since "dynamic" appears in the text, and the query finally returns true (many times). With dynamic cut, the computation of the first alternative stops after the first fail of part "love" "..." and the query readily returns only one true, as expected.

Example 3
This last example, presented in Figure 3, combines both kinds of dynamic cuts presented above. Function palindrome detects when a string X is a palindrome, word detects strings built only from letters, and palinWord indicates if its argument W is both a palindrome and a word. Thus palinWord "refer" returns true, while palinWord "!!!" returns false repeated three times. In this case both the *or* branch of the && function and the (possibly) repeated existential search in *palindrome* contribute to decrease the efficiency of the program. Observe that the use of the parallel and (&&) in this example cannot be easily replaced by the usual *sequential and*:

```
and true  Y = Y
and false Y = false
```

because this function requires the evaluation of the two boolean expressions, but palindrome either returns true or fails without returning false. Therefore a goal like palinWord "123" would fail with the sequential *and* but returns false when introducing the parallel *and*.

```
rev []          = []
rev [X|Xs]      = (rev Xs)++[X]

palindrome X = true <== Z ++ (rev Z) == X
palindrome X = true <== Z ++ [C] ++ (rev Z) == X

word []         = true
word [X|Xs]     = (isLetter X) && (word Xs)

isLetter X      = (ord(X)>=ord('a')) && (ord(X)<=ord('z'))

palinWord W = palindrome W && word W
```

Fig. 4. Palindrome Words

Some of the previous examples involve functions, like part, matches or palindrome, that return true or fail. This corresponds closely to the logic programming style. It could be argued that truly boolean functions returning true or false (instead of failure) are preferable in practice, since in (standard) FLP one cannot make computational use of failure; for instance, one cannot use the dicotomy true/failure to distinguish cases in a definition. But sometimes the true-valued version of a function is easier to define than the {true,false}-valued version; if the false value for that function is not used in the program then the simpler version is still useful. For instance consider the function part, defined in the program of Figure 1 (rule ($R3$)) as

$$\mathsf{part\ X\ Y = true <==U ++ X ++ V == Y}$$

This rule cannot be converted into a {true,false}-valued function simply replacing the body true by the condition:

$$\mathsf{part'\ X\ Y = U ++ X ++ V == Y}$$

Indeed, a goal like part' "time" "Once upon a time" will produce incorrect results:

```
false
more solutions? y
true ...
```

This is due to the presence of the extra variables U, V in the body of part'. In the first answer the substitution $\{\mathsf{U} \mapsto []\}$ is obtained and the strict equality

$$\mathsf{[] ++ "time" ++ V == "Once\ upon\ a\ time"}$$

returns false for any possible value of V, while in the second answer the substitution $\{\mathsf{U} \mapsto \text{"Once upon a "}, \mathsf{V} \mapsto []\}$ leads to

"Once upon a " ++ "time" ++ [] == "Once upon a time"

which yields the (correct) solution true. This, obviously, does not mean that part cannot be transformed into a {true,false}-valued function. A possible definition could be:

```
part' [] []         = true
part' [X|Xs] []     = false
part' [X|Xs] [Y|Ys] = (prefix [X|Xs] [Y|Ys]) or (part' [X|Xs] Ys)
```

with suitable definitions for functions prefix and or. Notice that this {true,false}-valued version of part is also a semantically deterministic function. It is even a non-ambiguous function in the sense of Def. 1 of next section, provided that prefix and or are non-ambiguous, as they indeed should be if they are defined in a natural way. Therefore, dynamic cut techniques could also be applied to this case, avoiding the repeated answers if we define or using a 'parallel' definition analogous to that of (&&). A similar discussion can be applied to the function palindrome of Figure 3.

Furthermore, there are extensions of the FLP paradigm [18, 17] where failure becomes a computationally useful construct, allowing in particular to 'complete' the true-valued version of a function with a *default* rule giving the value false.

4 Detecting Deterministic Functions

As we have seen, the deterministic nature of functions plays an important role when determining if a dynamic cut can be performed, as was illustrated in the examples above. We say that a function $f \in FS^n$ is *(semantically) deterministic* if for all ground terms $t_1 \ldots t_n$ the goal $f\ t_1 \ldots t_n$ cannot produce different data values. The functions ++, &&, part, matches,rev, palindrome, isLetter and palinWord of Section 3 are all deterministic, while the function maybe is not.
Now we introduce an adaptation of the non-ambiguity conditions in [15], which can serve as an easy mechanism for the effective recognition of deterministic functions. Despite their simplicity, these conditions are enough in most practical cases, in particular for the examples of Section 3.

Definition 1 (Non-ambiguous functions).
Let P be a program defining a set of functions G. We say that $F \subseteq G$ is a set of non-ambiguous functions *if all $f \in F$ verifies:*

(i) If $f(\overline{t}) = e \Leftarrow C$ is a defining rule for f, then $var(e) \subseteq var(\overline{t})$ and all function symbols in e belong to F (that is, extra variables *and ambiguous functions cannot occur in bodies).*

(ii) For any pair of variants of defining rules for f, $f(\overline{t}) = e \Leftarrow C$, $f(\overline{t'}) = e' \Leftarrow C'$, one of the following two possibilities holds:
   (a) Heads do not overlap, that is, $f(\overline{t})$ and $f(\overline{t'})$ are not unifiable.
   (b) If θ is a mgu of $f(\overline{t})$ and $f(\overline{t'})$, then $e\theta \equiv e'\theta$.

The second part of the definition is equivalent to say that the set of unconditional parts of defining rules for functions $f \in F$ is a weakly orthogonal TRS [6].

Not all the deterministic functions are non-ambiguous. However, the non-ambiguity criterion is enough to ensure that the cuts will be safe. A finer characterization of semantically deterministic functions would increase both the number of functions that can include dynamic cut and the number of cuts performed during the computations.

Remarks:

- The definition above characterizes determinism of the set of functions F as a whole. This is done so because the value of a function might depend on other functions, and in general the dependence can be mutual. In practice, recognition of deterministic functions can be done in a hierarchical way, by considering blocks of functions (in most cases, blocks will be singletons) depending on themselves or in other functions previously proved to be deterministic.

- Even if a function f, defined by a set of rules R_f, is non deterministic, it might happen that a subset of R_f define a deterministic function. This allows us to extend dynamic cuts to the deterministic subsets of non-deterministic functions.

- Notice that predicates defined in a pure logic program become non-ambiguous functions if the usual translation is made, which converts a clause with the form $h :- b_1, \ldots, b_n$ into the rule $h = true \Leftarrow b_1 == true, \ldots, b_n == true$.

Next we introduce two claims which establish the conditions for introducing the dynamic cut safely. The claims are stated in a hopefully precise but informal way, and they are also justified only at an informal level through some comments following the claims, and through the examples of this section and the previous one. A formal technical treatment of the issues discussed here is out of the scope of this paper.

Claim 1

Let G be a goal, f a deterministic function and e an expression of the form $e \equiv f(e_1, \ldots, e_n)$. If a computation of a head normal form for e occurs during the computation of G, and succeeds without:

(i) Binding any variable in e.
(ii) Computing a hnf for any expression $g(e'_1, \ldots, e'_m)$ where g is non-deterministic.

Then any other alternative to the computation of this hnf for e can be discarded, since it cannot contribute to produce a different answer for the original goal.

Remember that in our setting the answers for a goal are pairs of the form $(result, substitution)$. Condition (i) indicates that the substitution obtained for subgoal e is the identity and therefore every possible re-evaluation will produce a more particular substitution. The condition (ii) ensures that the computation has been produced using only deterministic functions and therefore the expression e cannot produce a different $result$.

For instance, this claim justifies the introduction of the dynamic cut in the case of the parallel && introduced as first example in Section 3.

Claim 2

Let G be a goal and $e \equiv f(e_1, \ldots, e_n)$ an expression, such that the computation of a hnf of e occurs during the computation of G. Let $f(\overline{t}) = r \Leftarrow C$ be a defining rule for the (possibly non-deterministic) function f used to compute such hnf. Then, if the condition C is successfully computed without:

(i) Binding any variable in $e_1, \ldots, e_n$, e.
(ii) Computing a hnf for any expression $g(e'_1, \ldots, e'_m)$ where g is non-deterministic.

Then any alternative re-evaluation of C can be discarded, since it cannot contribute to produce a different answer for the original goal.

Conditions *(i)* and *(ii)* are required for the same reasons given in Claim 1. Examples of situations corresponding to Claim 2 have been given in the examples 2 (for the function `part`) and 3 (for the function `palindrome`) of Section 3. However in both cases the involved functions were indeed deterministic. Figure 5 presents an additional example to illustrate the case with a non-deterministic function. In this program f, and therefore g, are non-deterministic functions, while h and p are deterministic.

```
f X = X                          p 0 0 = true
f X = X + 1                      p 0 1 = true
g X = f X <== p X Y == true      h 1   = 1
```

Fig. 5. Example for Claim 2

A goal like h (g 0) returns, without dynamic cut, the expected answer 1 repeated two times. The reason of this repetition is that solving h (g 0) requires first to evaluate g 0 to hnf; and this requires to check the condition p 0 Y == true. The condition succeeds in a first alternative giving the binding Y/0, but there is a second successful alternative for the condition giving the binding Y/1 which produces the second evaluation of g 0 to (the same) hnf and yields the repeated answer 1. What claim 2 states is that, since Y is local to the condition, this alternative can be pruned in case of future backtracking. Coming back to the goal in the example, after checking the condition, f 0 must be evaluated. A first obtained value is 0, which is then a first hnf for g 0. But 0 does not match the term 1 of the rule for sf h, and then backtracking is required. A second alternative for f 0 gives 0+1, which evaluates to 1. This matches the rule for h and we finally obtain the expected value 1. Notice the importance of cutting just after checking the condition, and not at the end of the application of the rule.

The latter would have pruned the use of the second rule for f when evaluating f 0, and therefore we would have missed the expected answer.

Notice that the condition (ii) in both claims 1 and 2 avoids the introduction of dynamic cuts even if the evaluation of a non-deterministic function has no influence over the answer obtained for e. The program in Figure 6 exemplifies why this condition is not too restrictive.

```
incNat X   = X + 1 <== natural X
natural X  = true   <== X>=0
zeroOrOne = 0
zeroOrOne = 1
```

Fig. 6. Example for Claim 2

In this program function zeroOrOne is non-deterministic while both incNat and natural are deterministic.

In principle we could think that, when computing a hnf for an expression $e \equiv$ natural e', the function natural could safely cut if no binding of variables is produced during the evaluation of its condition $X >= 0$, because regardless of whether some non-deterministic function has been evaluated or not, the function natural is going to produce *true* as result if the condition has been successfully solved. But we must remember that the computation of e can be a subcomputation of some initial goal, and therefore the evaluation of its condition can have some 'long distance' effects due to the mechanism of suspensions and sharing, as the following example shows.

Consider for instance the goal incNat zeroOrOne. The expected answers are 1 and 2 since the two values produces by zeroOrOne are natural numbers. To obtain these results, incNat must evaluate its condition natural X which in this computation would be natural zeroOrOne. Now function natural has itself a condition which will be zeroOrOne >= 0. To solve the condition the non-deterministic function zeroOrOne is evaluated first to 0, which satisfies the condition and produces the first result 1 for the goal (obtained by evaluating the right-hand side $0 + 1$ of incNat). In a similar way, by re-evaluating the condition of incNat the condition of natural will be in turn re-evaluated and the second value 2 will be produced.

But if natural had introduced a cut after the first evaluation of its condition it would have avoid the re-evaluation of zeroOrOne and therefore the second answer for the goal would have been *missed*.

Although following claims 1 and 2 we could safely introduce dynamic cut associated to many evaluations of hnf, most of these cuts would be unnecessary. Instead, we will include code for dynamic cut only in the two situations presented

in the examples of Section 3 and described precisely in the code generation of the next section.

5 A Prolog Implementation of Dynamic Cut

In this section we explain how to accommodate dynamic cut into the translation scheme $\mathcal{TOY} \rightarrow$ *Prolog* described in Section 2.2. We think that it is not difficult to extend the approach to other translation schemes, as far as they use Prolog's depth first search and backtracking. But it is not so clear how to include in an effective way dynamic cut in an operationally complete implementation where choices are evaluated in parallel.

Coming back to the scheme of Section 2.2, only the third phase of the translation, the code generation, is modified to introduce the dynamic cuts.

For this modification we will use a pair of auxiliary predicates:

- *varlist(E, Vs)*, which returns in *Vs* the list of variables occurring in E, taking into account the following criterion for collecting variables inside suspensions:
(1) If E contains an unevaluated suspension $susp(f(e_1, \ldots, e_n), R, S)$ and f is a non-deterministic function then R must be added to *Vs*. This is directly related to part *(ii)* of Claims 1 and 2, and it is essential for performing dynamic cut safely, as shown in the examples above.
(2) If E contains an evaluated suspension $susp(f(e_1, \ldots, e_n), R, S)$, then we proceed recursively collecting variables in R.
- *checkvarlist(Vs)*, which checks that all elements in *Vs* are indeed different variables. This ensures that no variable in *Vs* was bound during the evaluation of E.

The combination of *varlist* and *checkvarlist* in a code sequence like

$$\mathsf{varlist(E,Vs)},\ <\textit{compute something with } E>,\ \mathsf{checkvarlist(Vs)}$$

is an easy way of controlling that no variables in E have been bound during the computation. In many practical cases *Vs* will be empty, and then *checkvarlist(Vs)* is a trivial test. The actual implementation of *varlist* and *checkvarlist* is straightforward and can be found at `http://babel.dacya.ucm.es/rafa/cut`.
As we did in section 2.2, we distinguish cases according to the shape of the root of the definitional tree dt of a given function f:
Case 1 (the root is a **case** node):

Assume $dt \equiv f(\overline{s}) \rightarrow \mathbf{case}\ X\ \mathbf{of}\ \langle c_1 :\ dt_1 \ldots c_m :\ dt_m \rangle$
In this case different branches correspond to incompatible cases in a given position, and therefore there is nothing to prune. The generated code in this case is the same as if dynamic cut is not taken into account:

$$\begin{aligned} prolog(g, dt) = \{\ &\mathsf{g}(\overline{s},\mathsf{H})\ \mathsf{:-}\ \mathsf{hnf(X,HX)},\ \mathsf{g'}(\overline{s}\sigma,\mathsf{H}).\}\ \cup \\ &prolog(g', dt_1)\ \ldots \cup prolog(g', dt_m) \end{aligned}$$

where $\sigma = \mathsf{X/HX}$ and g' is a new function symbol.

Case 2 (the root is an **or** node):

Assume $dt \equiv f(\overline{s}) \rightarrow$ **or** $\langle dt_1 \mid \ldots \mid dt_m \rangle$
In this case, some of the (head of) rules in different branches might overlap, maybe yielding to different computations with the same result. Code for dynamic cut at the root can be useful, but it is safe only in case that the function defined by the tree is deterministic; otherwise, different branches, even overlapping, might produce different results and none of which should be pruned. To be precise: let R be the set of program rules in the leaves of dt. We consider two cases:

Case 2.1 If R defines a non-deterministic function, then code for dynamic cut cannot be added, and the code generated is the same we described in section 2.2:

$$\begin{aligned} prolog(g, dt) = \; & \{\ \mathsf{g}(\overline{s},\mathsf{H})\ \mathsf{:-}\ \mathsf{g}_1(\overline{s},\mathsf{H}).\} \cup \ \ldots \cup \ \{\ \mathsf{g}(\overline{s},\mathsf{H})\ \mathsf{:-}\ \mathsf{g}_m(\overline{s},\mathsf{H}).\} \cup \\ & prolog(g_1, dt_1) \cup \ \ldots \ \cup prolog(g_m, dt_m) \end{aligned}$$

where $g_1, \ldots, g_m$ are new function symbols.
Case 2.2 If R defines a deterministic function, then we add code for dynamic cut by adding a new auxiliary predicate g_{aux}:

$$\begin{aligned} prolog(g, dt) = \; & \{\ \mathsf{g}(\overline{s},\mathsf{H})\ \mathsf{:-}\ \mathsf{varlist}(\overline{s},\mathsf{Vs}), \\ & \qquad \mathsf{g}_{aux}(\overline{s},\mathsf{H}), \\ & \qquad (\mathsf{checkvarlist}(\mathsf{Vs}), \\ & \qquad !\ \%\ \textit{this is the dynamic cut} \\ & \qquad ; \\ & \qquad \mathsf{true}).\ \} \ \cup \\ & \{\mathsf{g}_{aux}(\overline{s},\mathsf{H})\ \mathsf{:-}\ \mathsf{g}_1(\overline{s},\mathsf{H}).\} \ \cup \ \ldots \ \cup \ \{\mathsf{g}_{aux}(\overline{s},\mathsf{H})\ \mathsf{:-}\ \mathsf{g}_m(\overline{s},\mathsf{H}).\} \ \cup \\ & prolog(g_1, dt_1) \ \cup \ \ldots \cup prolog(g_m, dt_m) \end{aligned}$$

where $g_{aux}, g_1, \ldots, g_m$ are new function symbols. Observe that g_{aux} is defined as g in the case 2.1, that is, as g would be defined without dynamic cut. The behavior of the clause for g is then clear: we collect the relevant variables of the call, and use g_{aux} to do the reduction; if after succeeding no relevant variable has been bound, we cut to prune other (useless) alternatives for g_{aux}.

The cut here is safe because we are in the conditions of the claim 1 of section 4: no variable have been bound and no non-deterministic function has been used because it would have modified its suspension variable, and checkvarlist would have failed.

We remark that, according to Prolog standard [10], the occurrence of ! in the clause for g above is indeed visible in the clause, and has therefore the desired effect.
Notice also that the condition required to add code for dynamic cut is *local* to the tree: only the rules in the tree are taken into account. This allows a 'fine tuning' of dynamic cut, which can be added to 'deterministic parts' of a function definition, even if the function is non-deterministic.
Case 3 (the tree is a leaf **try**):

Assume $dt \equiv \mathbf{try}\ R$, where R is a program rule

$$f(\overline{s}) \ = \ e \Leftarrow l_1 == r_1, \ldots, l_n == r_n$$

In this case it is always possible to add code for dynamic cut between the code for the conditions and the code for the body e. Some care must be taken with the variables in the conditions not occurring in the head $f(\overline{s})$. If any of these extra variables does not occur in the body e, then it is an existential variable, whose only role is to witness the condition. The relevant fact is that if the conditions succeed with some bindings for existential variables, there is no need of finding alternative bindings for such variables. But if one extra variable of the conditions occurs also in e, it might contribute to its value, and therefore to the value of $f(\overline{s})$; this means that bindings for such variables must inhibit the dynamic cut. To take this into account is quite easy: just add the variables in e to the list of variables relevant for dynamic cut.

$prolog(g, dt) =$
```
{ g0(s̄,H) :- varlist((s̄, e),Vs), % notice the body e
             equal(l1,r1), ..., equal(ln, rn),
             (checkvarlist(Vs),
             ! % this is the dynamic cut
             ; true),
             hnf(e,H). }
```

In this case the cut fulfills the conditions of the claim 2 of section 4. We remark that this code is correct even if the body e is non-deterministic, because the cut is placed before evaluating the body, which implies that we only cut the re-evaluation of the conditional part of the rule.

5.1 Examples

Here we present a few examples of translations into Prolog following the ideas commented above. The complete generated code for the examples can be found at

`http://babel.dacya.ucm.es/rafa/cut`

It is worth noticing that the code found there is not *exactly* the code described in the paper: apart from typical optimizations, as the real code is going to be executed within $\mathcal{TOY}$, it must take into account disequality constraints [5, 1], which are embedded in the system.

Parallel and Since the function && is deterministic and has an **or** node at the root of its definitional tree, dynamic cut code is added for it. Since the rules are unconditional, **try** nodes do not require dynamic cut. The Prolog code for && is then:

```
&&(X,Y,H) :-
        varlist((X,Y),Vs),
        &&_aux(X,Y,H),
        (checkvarlist(Vs),
        ! % this is the dynamic cut
        ; true).
. . . . . .
```

where the auxiliary predicate $\&\&_{aux}$ is defined exactly as would be && without the dynamic cut.

Simple queries In this example, the functions part and matches accept dynamic cut, the first because its rule has a condition with existential variables, and the second because it is deterministic and has an **or** node in its definitional tree.

The code for part including dynamic cut is:

```
part(X,Y,H) :-
        varlist((X,Y),Vs),
        equal(susp(++(U,susp(++(X,V),R,S)),R',S'), Y),
        (checkvarlist(Vs),
        !    % dynamic cut after the conditions
        ; true),
        hnf(true,H).
```

The code for match is now:

```
matches(X, Text, H) :-
        hnf(X, HX),
        matches_1(HX, Text, H).
% the three possibilities of the 'case' branch
matches_1(single(S), Text, H) :-
        equal(susp(part(S, Text), R, Flag),true)
        hnf(true,H).
matches_1(and(S, S'), Text, H) :-
        equal(susp(matches(S, Text), R_1, Flag_1),true),
        equal(susp(matches(S', Text), R_2, Flag_2),true),
        hnf(true,H).
matches_1(or(S,S'), Text, H) :-
        varlist(or(S,S'),Vs),
        matches_aux(or(S,S'), Text, H).
        (checkvarlist(Vs),
        ! % this is the dynamic cut
        ;
        true).
matches_aux(or(S,S'), Text, H) :-
        matches_1'(or(S,S'), Text, H).
```

```
matches_aux(or(S,S'), Text, H) :-
      matches_1"(or(S,S'), Text, H).
% the two possibilities of the 'or' branch
matches_1'(or(S,S'), Text, H) :-
      equal(susp(matches(S, Text), R, Flag),true),
      hnf(true,H).
matches_1"(or(S,S'), Text, H) :-
      equal(susp(matches(S', Text), R, Flag),true),
      hnf(true,H).
```

Notice that the only differences between this code and the version without dynamic cut presented in section 2.2 are the renaming of the third clause of matches$_1$ to matches$_{aux}$ and the introduction of the clause with the dynamic cut.

6 Experimental results

Fig. 7 presents some experimental results obtained with the system $\mathcal{TOY}$[3]. In addition to the examples of Section 3 we have used two examples:
- *graph.toy*: This program defines a graph with the shape of a grid, where each node is connected to its nearest right and down nodes. Also, a function to check whether two nodes are connected is defined. The natural coding of this function includes an existential condition in a program rule that will include code for the dynamic cut.
- *composite.toy*: Program to check whether a number is composite, i.e. not prime. This is achieved by looking for two numbers whose product is the desired number, and this, again, is naturally represented in FLP languages by an existential search in the condition of a program rule. The dynamic cut will stop the computations after the first decomposition is found if the number is not prime.
In the following we describe briefly each goal.
- G_1 is false && (false && (.... (false && false) ...)) == true with 100000 false values.

- G_2 is (...((false && false) && false) && ...) && false == true with 5000 false values. In contrast to G_1, in this case dynamic cut is not really effective, and the code including cut is indeed slightly worse due to the run-time checking of bindings. This behaviour is due to the order of rules (of &&) and evaluation, because in G_1 the 'expensive' branch (which evaluates the secons conjunct (false && (.... (false && false) ...)) to obtain false) is pruned, while in G_2 what is pruned is the 'cheap' one (which evaluates false to obtain false).

- G_3 is matches (or (and (single "cut") (single "love")) (single "dynamic")) intro where intro represents the text of the introduction of this paper.

- G_4 is matches (and (and (single "is") (single "this")) (single "love?")) intro.

[3] Running on a PC under O.S. Linux with processor Intel Celeron at 600 MHZ and 128 Mb RAM.

Program	Goal	Without Dynamic Cut	With Dynamic Cut
example1.toy	G1	3.4 sec.	0 sec.
example1.toy	G2	105.2 sec.	119.8 sec.
example2.toy	G3	30.7 sec.	2.5 sec.
example2.toy	G4	327.3 sec.	2.3 sec.
example2.toy	G5	>5 hours	2.0 sec.
example3.toy	G6	33.5 sec.	4.8 sec.
graph.toy	G7	64.2 sec.	0 sec.
graph.toy	G8	>5 hours	0 sec.
graph.toy	G9	>5 hours	0.1 sec.
graph.toy	G10	66.7 sec.	70.6 sec.
composite.toy	G11	151.0 sec.	0.4 sec.
composite.toy	G12	>5 hours	4.0 sec.
example1.toy	G13	loops	0 sec.

Fig. 7. Runtime Table

- G_5 is matches (and (and (single "is") (single "a")) (single "love")). In this example notice that the goal fails due to the lack of "love" in the introduction, but both "a" and "is" occur many times in the text and therefore the search space is really huge.

- G_6 is palinWord "11...11" with "11..11" representing the string with 200 repetitions of digit 1 (which is obviously a palindrome but not a word).

In the rest of the examples the goals have been forced to fail in order to check the time required to examine the whole search space. This is not as artificial as it could seem; on the contrary it happens whenever the goal is evaluated as part of a subcomputation that finally fails.
- G_7 looks for a path between the upper-left and the lower-right corner of a grid of 10×10 nodes. Without dynamic cut the backtracking will try all possible paths in the graph, but the dynamic cut stops after finding the first successful path. G_8 and G_9 are analogous to the previous goal but for grids of 20×20 and 100×100, respectively.

- G_{10} looks for paths from the upper-left corner to a generic node represented as a variable N. In this case the cut takes no effect because variable N is bound during the computations and cutting would not be safe, and the times with and without dynamic cut are similar.

- G_{11} checks if number 1000 is not prime, while G_{12} is analogous but for number 10000.

- G_{13}. In addition to these examples it is easy to find goals where dynamic cut avoids non-termination. For instances $G_{13} \equiv$ false && loop == true, where loop is defined as loop = loop, fails with dynamic cut while loops without it.

7 Conclusions

This paper presents a mechanism of *dynamic cut* for lazy FLP programs that can be easily introduced in a Prolog-based implementation. The technique requires a static analysis of determinism and the modification of the segment of the generated code where the cut is feasible (deterministic functions with *or* branches and rules with existential conditions).
By including dynamic cuts, the efficiency of several computations both in terms of time and space is improved, often dramatically. This is done by avoiding redundant non-deterministic computations related to the evaluation of semantically deterministic functions. In contrast to Prolog cuts, the dynamic cut proposed here is transparent to the programmer (since it is automatically introduced by the system in the generated code) and it is safe.

The second consequence of the cut is that many repeated answers can be avoided. Also non-terminating computations become, in some cases, terminating. However, dynamic cut does not change the set of computed answers.

Because of these two benefits, functions that are usually avoided in FLP, like the *parallel and* presented in Figure 3, can now be used without decreasing the efficiency of the computations.

Compared to a previous work ([15]) on this subject our proposal presents three major improvements:
• Non-deterministic functions are considered.
• The introduction of the dynamic cut is related to definitional trees allowing the integration of the technique into current systems based on demand driven strategies.
• We show how to incorporate the technique in systems that generate code by transforming FLP programs into Prolog-code. This, together with the two previous points, makes the technique fully applicable to several FLP implementations.

In principle the techniques discussed in this paper can be applied to pure logic program, since any pure logic program is non-ambiguous in the sense of definition 1. However, including dynamic cuts could produce undesirable effects in the real practice of logic languages such as Prolog, where many non-logical features are used. Consider for instance the usual definition of repeat, a predicate commonly used in Prolog to implement a failure loop to be combined with side-effects:

```
repeat.
repeat :- repeat.
```

With dynamic cut the second clause of repeat would never be used, thus converting repeat into a useless predicate. We think that this kind of problems explains

why other theoretical proposals in the field of logic programming [9, 8] are not usually implemented in real logic systems.

As future work, we plan to fully integrate the optimization in the system $\mathcal{TOY}$ and to improve the implementation of the mechanism used to detect whether a relevant variable has been bound. The present proposal has proved to be effective, but it is rather naive, since it requires complete traversals of expressions and lists of variables. In a different line, a deeper theoretical work would be desirable which would both extend the class of functions qualified as deterministic and provide an operational framework suitable to prove the properties of the technique.

Acknowledgement

We thank the anonymous referees for many valuable comments.

References

1. M. Abengózar-Carneros, P. Arenas-Sánchez, R. Caballero, A. Gil-Luezas, J.C. González-Moreno, J. Leach-Albert, F.J. López-Fraguas, M. Rodríguez-Artalejo, J.J. Ruz-Ortiz and J. Sánchez-Hernández. $\mathcal{TOY}$: *A Multiparadigm Declarative Language. Version 1.0.* Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Tech. Report SIP-119/00, February 2002. Available at http://titan.sip.ucm.es/toy/toyreport.pdf.
2. S. Antoy. *Definitional Trees.* In Proc. Int. Conf. on Algebraic and Logic Programming (ALP'92), Springer LNCS 632, 1992, 143-157.
3. S. Antoy. *Constructed-based Conditional Narrowing.* In Proc. Int. Conf. on Principles and Practice of Declarative Programming (PPDP'01), ACM Press, 2001, 199-206.
4. S. Antoy, R. Echahed and M. Hanus. *A Needed Narrowing Strategy.* Journal of the ACM Vol. 47, no. 4, 2000, 776-822.
5. P. Arenas-Sánchez, A. Gil-Luezas and F. J. López Fraguas: *Combining Lazy Narrowing with Disequality Constraints*, In Proc. Int. Symp. on Programming Languages Implementation and Logic Programming (PLILP'94), Springer LNCS 844, 1994, 385-399.
6. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
7. P.H. Cheong and L. Fribourg. *Implementation of narrowing: The Prolog-based approach.* In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, Logic programming languages: constraints, functions, and objects, The MIT Press, 1993, 1-20.
8. P. Codognet and T. Sola. *Extending the WAM for Intelligent Backtracking.* In Proc. Int. Conf. on Logic Programming (ICLP'91), The MIT Press, 1991, 127-141.
9. S.K. Debray and D.S. Warren. *Functional Computations in Logic Programs.* ACM Transactions on Programming Language Systems, 11(3), 1989, 451-481.
10. P. Deransart, A. Ed-Dbali and L. Cervoni. *Prolog: The Standard.* Springer, 1996.
11. J. C. González-Moreno. *A Correctness Proof for Warren's HO into FO Translation.* Procs. of GULP'93, 1993, 569-585.

12. J.C. González-Moreno, T. Hortalá-González, F.J. López-Fraguas and M. Rodríguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic*. J. of Logic Programming, 40(1), 1999, 47-87.
13. M. Hanus. *Curry: An Integrated Functional Logic Language*. Version 0.7.1, June 2000. Available at http://www.informatik.uni-kiel.de/curry/report.html.
14. R. Loogen, F.J. López-Fraguas, and M. Rodríguez-Artalejo. *A Demand Driven Computation Strategy for Lazy Narrowing*. In Proc. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'93), Springer LNCS 714, 1993, 184-200.
15. R. Loogen and St. Winkler. *Dynamic Detection of Determinism in Functional-Logic Languages*. In Proc. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'91), Springer LNCS 528, 1991, 335-346.
16. F.J. López-Fraguas, and J. Sánchez-Hernández. *$\mathcal{TOY}$ a Multiparadigm Declarative System*. In Proc. Int. Conf. on Rewriting Techniques and Applications (RTA'99), Springer LNCS 1631, 1999, 244-247.
17. F.J. López-Fraguas and J. Sánchez-Hernández. *A proof theoretic approach to failure in functional logic programming*. To appear in *Theory and Practice of Logic Programming*.
18. J.J. Moreno-Navarro. *Default rules: An extension of constructive negation for narrowing-based languages*. In Proc. Int. Conf. on Logic Programming (ICLP'94), The MIT Press, 1994, 535-549.
19. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. *Logic Programming with Functions and Predicates: The Language BABEL*. J. of Logic Programming, 12, 1992, 191-223.
20. S. Narain. *Optimization by Non-Deterministic, Lazy Rewriting*. In Proc. Int. Conf. on Rewriting Techniques and Applications (RTA'89), Springer LNCS 355, 1989, 326-342.
21. D.H.D. Warren. *Higher-order extensions to Prolog: are they needed?*. Hayes J.E., Michie D. and Pao Y-H. (eds), Machine Intelligence 10, Ellis Horwood, 1982, 441-454.