

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, both showing 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 to 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 huge search spaces with their associated overheads both in terms of time and space.

In their work [10], Rita Loogen and Stephan Winkler presented a technique for the run-time detection of deterministic computations that can be used to safely prune the search space. This technique is known as *dynamic cut*. Unfortunately the programs considered in this work did not include non-deterministic functions, which are used extensively in FLP nowadays. Also the implementation (based on a modification of an abstract machine) did not follow the demand driven strategy [4, 11], which in the meantime has been adopted by all the current implementations of FLP languages.

Our proposal adapts the original idea 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

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

adaptable to the current implementations of FLP languages based on translation into Prolog code. The result of implementing the dynamic cut are 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 no formal way.

In the following section we introduce some preliminaries. Section 3 discusses several examples motivating the introduction of the dynamic cut in the implementation of FLP programs. Section 4 introduces more formally the key concept of deterministic function, while Section 5 discusses an implementation of the technique. 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 Preliminaries

All the examples are written in the concrete syntax of the lazy FLP language \mathcal{TOY} [12] but can be easily adapted to other FLP languages like Curry [9]. 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. *Partial expressions* $e \in Exp_{\perp}$ have the syntax

$$e ::= \perp \mid X \mid h \mid (e \ e') \mid (e_1, \dots, e_n)$$

where $X \in Var$, $h \in DC \cup FS$. Expressions of the form $(e \ e')$ stand for the application of expression e (acting as a function) to expression e' (acting as an argument), while expressions (e_1, \dots, e_n) represent tuples with n components. As usual, we assume that application associates to the left and thus $(e_0 \ e_1 \dots e_n)$ abbreviates $((\dots (e_0 \ e_1) \dots) e_n)$. The set of data variables occurring in e is written $var(e)$. *Partial patterns* $t \in Pat_{\perp} \subset Exp_{\perp}$ are built as

$$t ::= \perp \mid X \mid c \ t_1 \dots t_m \mid f \ t_1 \dots t_m \mid (t_1, \dots, t_n)$$

where $X \in Var$, $c \in DC^n$, $0 \leq m \leq n$, $f \in FS^n$, $0 \leq m < n$ and t_i partial patterns for all $1 \leq i \leq m$. They represent *approximations* of the values of expressions. Expressions and patterns without any occurrence of \perp are called *total*. We write Exp and Pat for the sets of total expressions and patterns, respectively.

The *approximation ordering* \sqsubseteq is defined as the least partial ordering over Pat_{\perp} satisfying the following properties:

- $\perp \sqsubseteq t$, for all $t \in Pat_{\perp}$.
- $h \ \bar{t}_m \sqsubseteq h \ \bar{s}_m$ whenever these two expressions are patterns and $t_i \sqsubseteq s_i$ for all $1 \leq i \leq m$.

A *total substitution* is a mapping $\theta : Var \rightarrow Pat$ with a unique extension $\hat{\theta} : Exp \rightarrow Exp$, which will be noted also as θ . The set of all substitutions is noted as $Subst$. The set of all the *partial substitutions* $\theta : Var \rightarrow Pat_{\perp}$ is denoted as $Subst_{\perp}$ and defined analogously. We define $dom(\theta)$ as the set of all variables X s.t. $\theta(X) \neq X$, and the *range* $ran(\theta)$ as $\bigcup_{X \in dom(\theta)} var(\theta(X))$. By convention, we write $e\theta$ instead of $\theta(e)$, and $\theta\sigma$ for the composition of θ and σ , such that $e(\theta\sigma) = (e\theta)\sigma$ for any e . Another useful notion is the *approximation* ordering over $Subst_{\perp}$, defined by the condition $\theta \sqsubseteq \theta'$ iff $\theta(X) \sqsubseteq \theta'(X)$, for all $X \in Var$.

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*:

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

where e_i, e'_i and r are expressions (that can contain new extra variables) and each t_j is a pattern, with no variable occurring more than once in different t_k, t_l .

Each function is assumed to have a *definitional tree* [2, 11] 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 by this modification is obviously equivalent and will be more suitable for our purposes.

We consider goals as expressions e and *answers* as pairs (t, σ) where t is a pattern representing a result obtained by evaluating e , while σ is a substitution of patterns 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: $e_1 == e'_1, \dots, e_k == e'_k$, simply by assuming that an auxiliary function: $\text{main } R_1 \dots R_n = \text{true} \Leftarrow e_1 == e'_1, \dots, e_k == e'_k$ is introduced with $\{R_1, \dots, R_n\} = vars(e_1) \cup vars(e'_1) \cup \dots \cup vars(e_k) \cup vars(e'_k)$, and then evaluating the goal $\text{main } R_1 \dots R_n$. 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.

The semantics of our programs is specified by the following calculus, presented in previous works (see [8] for instance) as a convenient framework for lazy functional-logic languages.

BT $e \rightarrow \perp$

RR $X \rightarrow X$ with $X \in Var$

DC $\frac{e_1 \rightarrow t_1 \dots e_m \rightarrow t_m}{h \bar{e}_m \rightarrow h \bar{t}_m} \quad h \bar{t}_m \in Pat_{\perp}$

JN $\frac{e \rightarrow t \quad e' \rightarrow t}{e == e'} \quad t \in Pat \text{ (total pattern)}$

FA $\frac{e_1 \rightarrow t_1 \dots e_n \rightarrow t_n \quad C \quad r \rightarrow s \quad s \bar{a}_k \rightarrow t}{f \bar{e}_n \bar{a}_k \rightarrow t} \quad (f \bar{t}_n \rightarrow r \Leftarrow C) \in [P]_{\perp}, \quad t \neq \perp$

In all the *SC* rules, $e, e_i \in Exp_{\perp}$ are partial expressions, $t_i, t, s \in Pat_{\perp}$ are partial patterns and $h \in DC \cup FS$. The notation $[P]_{\perp}$ in rule *FA* stands for the set $\{(l \rightarrow r \Leftarrow C)\theta \mid (l \rightarrow r \Leftarrow C) \in P, \theta \in Subst_{\perp}\}$ of partial instances of the rules from P . The labels of the different inference rules have the following intended meanings: *BT* stands for *Bottom*, *RR* for *restricted reflexivity*, *DC* for *decomposition*, *JN* for *joinability* and *FA* for *function application*. The expressions of the form $e \rightarrow t$ are called *approximation statements*. The idea is that $e \rightarrow t$ can be proved in the semantic calculus whenever t represents an approximation of e . In the following will use the notation $P \vdash_{SC} e \rightarrow t$ to denotate that $e \rightarrow t$ holds semantically w.r.t. the program P . Notice however that if e is a pattern then no application of rule *FA* is needed in the semantic proof and hence the program P is not relevant. In this case we will say simply that $e \rightarrow t$ holds.

Regarding the operational semantics, we assume that goals are solved by means of an operational mechanism based on needed narrowing with sharing [3, 4, 11], as well as a Prolog-based implementation as described in [11]. A main component of the operational mechanism is the computation of *head normal forms* (hnf) for expressions.

3 Motivating Examples

In this section we present examples showing informally the two situations where dynamic cut 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 additional ones can be found at <http://babel.dacya.ucm.es/cut>.

Example 1: Parallel and

Figure 1 shows a correct way of defining the *and* connective in FLP programming, known as *parallel and*, together with its definitional tree.

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

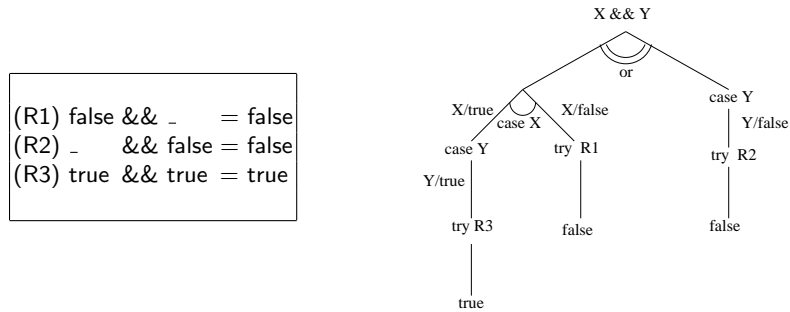


Fig. 1. The parallel and with its definitional tree

```

>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 [10]) as *semantically deterministic*, which means that if the first 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 [10], 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 $X==\text{false}$ or $Y==\text{false}$. 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 overload in the implementation.

Example 2

The second example shows the second type of dynamic cut. The program in Figure 2 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*. 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

```

infixr 50 ++
[] ++ Ys      = Ys
[X|Xs] ++ Ys  = [X|Xs ++ Ys]

part X Y      = true <== U ++ X ++ V == Y

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

matches (single S) Text = true <== part S Text
matches (and S S') Text = true <== matches S Text, matches S' Text
matches (or S S') Text  = true <== matches S Text
matches (or S S') Text  = true <== matches S' Text

```

Fig. 2. Simple Queries

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 combines both kinds of dynamic cuts presented above. Func-

```

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. 3. Palindrome Words

tion **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*.

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. Next we define precisely the concept of deterministic function. The initial idea is that a function f of arity n is deterministic if when it is applied to n ground terms t_1, \dots, t_n it can produce one value at most, i.e. there is at most one t such that $P \vdash_{SC} f\ t_1, \dots, t_n \rightarrow t$.

However in our setting this idea is not correct. The reason is that $f\ t_1, \dots, t_n \rightarrow t$ must not be read like an equality equation, i.e. like " t is $f\ t_1, \dots, t_n$ " but as " t is an approximation of $f\ t_1, \dots, t_n$ ", and $f\ t_1, \dots, t_n$ can have (and usually it will have) different approximations even for determinist functions f . For instance

$$\text{repeat } X = [X \mid \text{repeat } X]$$

should be deterministic. However, for example both $P \vdash_{SC} \text{repeat } 0 \rightarrow [0 \mid \perp]$ and $P \vdash_{SC} \text{repeat } 0 \rightarrow [0, 0 \mid \perp]$ hold.

The point is that in this case the terms t, t' must be themselves approximations of another term t'' which intuitively better represents the value of the function (in the case of *repeat* t'' could be for instance $[0, 0, 0 \mid \perp]$). This idea motivates the following definition, represented graphically in Fig. 4.

Definition 1 (Deterministic functions).

We say that a function $f \in FS^n$ is (semantically) deterministic if for all ground terms $t_1 \dots t_n$ such that $P \vdash_{SC} f\ t_1 \dots t_n \rightarrow t$ and $P \vdash_{SC} f\ t_1 \dots t_n \rightarrow t'$, with t, t' patterns, there exists another pattern t'' such that $t'' \rightarrow t$, $t'' \rightarrow t'$ and $P \vdash_{SC} f\ t_1 \dots t_n \rightarrow t''$.

Now we introduce an adaptation of the non-ambiguity conditions in [10], which can serve as an easy mechanism for the effective recognition of deterministic functions. Despite their simplicity, they are enough in most practical cases, in particular for the examples of section 3.

Definition 2 (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:

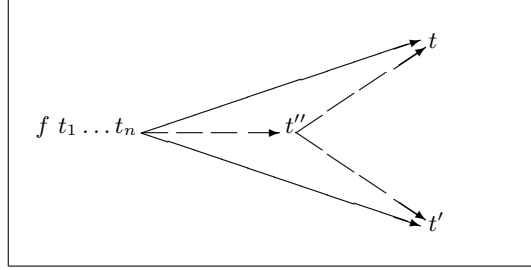


Fig. 4. Graphical interpretation of definition 1

- (i) If $f(\bar{t}) = e \Leftarrow C$ is a defining rule for f , then $\text{var}(e) \subseteq \text{var}(\bar{t})$ and all function symbols in e belong to F (that is, extra variables and possibly non deterministic functions cannot occur in bodies).
- (ii) For any pair of variants of defining rules for f , $f(\bar{t}) = e \Leftarrow C$, $f(\bar{t}') = e' \Leftarrow C'$, one of the following two possibilities holds:
 - (a) Heads do not overlap, that is, $f(\bar{t})$ and $f(\bar{t}')$ are not unifiable.
 - (b) Right-hand sides can be fused, that is, if θ is a mgu of $f(\bar{t})$ and $f(\bar{t}')$, 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 [5].

Next theorem relates definitions 1 and 2.

Theorem 1.

Non-ambiguous functions are semantically deterministic functions.

Proof

We prove the following, more general, result:

Let P a program, $e \in \text{Exp}_\perp$, and $t, t' \in \text{Pat}_\perp$ patterns such that $P \vdash_{SC} e \rightarrow t$ and $P \vdash_{SC} e \rightarrow t'$. Assume also that in the proof tree of these approximation statements all the function rules used correspond to non-ambiguous functions. Then there exists a pattern t'' s.t. $t'' \rightarrow t$, $t'' \rightarrow t'$, and $P \vdash_{SC} e \rightarrow t''$.

In the proof we use the following auxiliary lemmata:

Lemma 1

Let $t \in \text{Pat}_\perp$, $s \in \text{Pat}$ and $\theta \in \text{Subst}_\perp$. If $t \rightarrow s\theta$ then there exists $\theta' \in \text{Subst}_\perp$, $\theta \sqsubseteq \theta'$, such that $t = s\theta'$.

The idea is that if $t \rightarrow s\theta$ is because $s\theta$ has some occurrences of \perp at positions where t has some subterm $t_i \neq \perp$. Since s is total each value \perp comes from the substitution in θ of some variable X_i for a term t'_i that includes \perp . Then we can build a substitution θ' analogous to θ but replacing any occurrence of these \perp by the corresponding t_i .

Lemma 2

Let $t, t' \in Pat_{\perp}$, $e \in Exp_{\perp}$ and P a program. If $P \vdash_{SC} e \rightarrow t$ and $t \rightarrow s$ then $P \vdash_{SC} e \rightarrow s$.

Lemma 3

Let $t \in Pat_{\perp}$, $e \in Exp_{\perp}$, $\theta, \theta' \in Subst_{\perp}$, and P a program. If $P \vdash_{SC} e\theta \rightarrow t$ and $\theta \sqsubseteq \theta'$ then $P \vdash_{SC} e\theta' \rightarrow t$, with the depth of proof tree for $e\theta' \rightarrow t$ less than or equal to the depth of the proof tree for $e\theta \rightarrow t$.

Lemma 4

Let $t, s, s' \in Pat_{\perp}$, $a_1 \dots a_k, e \in Exp_{\perp}$ and P a program. If $P \vdash_{SC} s \overline{a_k} \rightarrow t$ and $s' \rightarrow s$ then $P \vdash_{SC} s' \overline{a_k} \rightarrow t$.

Lemma 5

Let $t, t' \in Pat_{\perp}$, and P a program. Then $t' \sqsubseteq t$ iff $P \vdash_{SC} t \rightarrow t'$

Lemma 6

Let $e, e' \in Exp_{\perp}$, $\theta, \theta' \in Subst_{\perp}$ and P a program. If $P \vdash_{SC} e\theta == e'\theta$ and $\theta \sqsubseteq \theta'$ then $P \vdash_{SC} e\theta' == e'\theta'$.

The proofs of these results are straightforward by using induction on the structure of the correspondent proof trees (analogous results can be found in [7]).

Now we are ready to start the proof.

If $t = \perp$ then it is enough to consider $t'' = t'$. Analogously if $t' = \perp$ then $t'' = t$. Otherwise we proceed by induction on the depth of the proof tree T for $e \rightarrow t$.

Basis. $depth(T) = 0$. Since $t \neq \perp$ there are two possibilities:

- $e = X, t = t' = X$ (X variable) and RR the applied rule.
- $e = c, t = t' = c$ (c constructor) and DC the applied rule.

In both cases the result holds for $t'' = t$.

Inductive Case. $depth(T) = n + 1$, and we assume that the result holds for all T' with $depth(T') \leq n$.

Depending of the form of e there are two possible semantic rules applied at the root:

- Rule DC . Then e is of the form $h \bar{e}_m$, and t, t' respectively $t \equiv h \bar{t}_m$ and $t' \equiv h \bar{t}'_m$, for some t_i, t'_i . Then the first inference step of the proofs for $e \rightarrow t$ and $e \rightarrow t'$ is:

$$\frac{e_1 \rightarrow t_1 \dots e_m \rightarrow t_m}{h \bar{e}_m \rightarrow h \bar{t}_m} \qquad \frac{e_1 \rightarrow t'_1 \dots e_m \rightarrow t'_m}{h \bar{e}_m \rightarrow h \bar{t}'_m}$$

Now, since for each $1 \leq i \leq m$ both $P \vdash_{SC} e_i \rightarrow t_i$ and $P \vdash_{SC} e_i \rightarrow t'_i$, and the depth of the proof tree of $e_i \rightarrow t_i$ must be less than or equal to n , by the inductive hypothesis exists t''_i such that $t''_i \rightarrow t_i$, $t''_i \rightarrow t'_i$, and $P \vdash_{SC} e_i \rightarrow t''_i$. Then by taking $t'' \equiv h \bar{t}''_m$ is straightforward to check that:

- $t'' \rightarrow t$ (by applying DC at the root and then remembering that $t''_i \rightarrow t_i$ for each i).

- $t'' \rightarrow t'$ analogously.
 - $P \vdash_{SC} e \rightarrow t''$ (by applying DC at the root and then using that $P \vdash_{SC} e_i \rightarrow t''_i$ for each i).
- Rule FA . Then $e = f \bar{e}_n \bar{a}_k$ and the first inference steps in the proofs of $e \rightarrow t$ and $e \rightarrow t'$ must be

$$\frac{e_1 \rightarrow s_1 \theta \quad \dots \quad e_n \rightarrow s_n \theta \quad C\theta \quad r\theta \rightarrow s \quad s \bar{a}_k \rightarrow t}{f \bar{e}_n \bar{a}_k \rightarrow t}$$

and

$$\frac{e_1 \rightarrow s'_1 \theta' \quad \dots \quad e_n \rightarrow s_n \theta' \quad C'\theta' \quad r'\theta' \rightarrow s' \quad s' \bar{a}_k \rightarrow t'}{f \bar{e}_n \bar{a}_k \rightarrow t'}$$

respectively, where $(f \bar{s}_n \rightarrow r \Leftarrow C)\theta$ and $(f \bar{s}'_n \rightarrow r' \Leftarrow C')\theta'$ are the program rules used in each case.

Now, for all $1 \leq i \leq m$ hold:

- $P \vdash_{SC} e_i \rightarrow s_i \theta$ and $P \vdash_{SC} e_i \rightarrow s'_i \theta'$
- The depth of the proof tree of $e_i \rightarrow s_i \theta$ must be less than or equal to n

Then, by the inductive hypothesis, exists s''_i such that $s''_i \rightarrow s_i \theta$, $s''_i \rightarrow s'_i \theta'$, and $P \vdash_{SC} e_i \rightarrow s''_i$.

Notice that the s_i, s'_i are all total patterns (only total patterns are allowed in the left-hand side of a function rule). Then, by Lemma 1 there exists θ_1, θ_2 such that: $\theta \sqsubseteq \theta_1$, $\theta' \sqsubseteq \theta_2$, $s''_i = s_i \theta_1$ and $s''_i = s'_i \theta_2$. Also, s_i and s'_i have no common variables, since each time a program rule is used all its variables are implicitly renamed to new variables, and hence can assume that θ_1, θ_2 are such that $s_i(\theta_1 \theta_2) = s'_i(\theta_1 \theta_2)$. Therefore by taking $\theta'' = \theta_1 \theta_2$ we have $s_i \theta \rightarrow s_i \theta''$ (from Lemma 5, because $s_i \theta'' = s_i \theta_1$ and $\theta_1 \sqsubseteq \theta$). By using now Lemma 2:

$$P \vdash_{SC} e_i \rightarrow s_i \theta'' \quad (1)$$

Therefore $\theta'' = \theta_1 \theta_2$ is an unifier of the left-hand sides of the two rules $f \bar{s}_n \rightarrow r \Leftarrow C$ and $f \bar{s}'_n \rightarrow r' \Leftarrow C'$, i.e. the two rules are overlapping, and by the definition 2 the equality $r\theta'' = r'\theta''$ must hold. But $r\theta'' = r\theta_1$ and $r'\theta'' = r'\theta_2$.

Then

- Since $P \vdash_{SC} r\theta \rightarrow s$ and $\theta \sqsubseteq \theta_1$, by Lemma 3, $P \vdash_{SC} r\theta_1 \rightarrow s$ and hence $P \vdash_{SC} r\theta'' \rightarrow s$
- Since $P \vdash_{SC} r'\theta' \rightarrow s'$ and $\theta' \sqsubseteq \theta_2$, by Lemma 3, $P \vdash_{SC} r'\theta_2 \rightarrow s'$. Using now the equality $r'\theta_2 = r'\theta'' = r\theta''$ we have $P \vdash_{SC} r\theta'' \rightarrow s'$

By using again the inductive hypothesis, there must exists s'' such that $s'' \rightarrow s$, $s'' \rightarrow s'$ and

$$P \vdash_{SC} r\theta'' \rightarrow s'' \quad (2)$$

Now since $s'' \rightarrow s$ and $s \bar{a}_k \rightarrow t$ by using Lemma 4 we have that $P \vdash_{SC} s'' \bar{a}_k \rightarrow t$ and analogously we obtain $P \vdash_{SC} s'' \bar{a}_k \rightarrow t'$ and by using again the inductive hypothesis there must exist t'' such that $t'' \rightarrow t$, $t'' \rightarrow t'$ and

$$P \vdash_{SC} s'' \bar{a}_k \rightarrow t'' \quad (3)$$

To finish the proof we only need to prove that t'' also satisfies $P \vdash_{SC} e \rightarrow t''$, i.e. that $f \bar{e}_n \bar{a}_k \rightarrow t''$. Since by hypothesis every atomic condition in $C_i \in C$ verifies $P \vdash_{SC} C_i \theta$, and $\theta \sqsubseteq \theta''$, it is easy to check by using Lemma 6 that

$$P \vdash_{SC} C_i \theta'' \quad (4)$$

for all $C_i \in C$

Now from (1), (2), (3), and (4) we have:

$$\frac{e_1 \rightarrow s_1 \theta'' \quad \dots \quad e_n \rightarrow s_n \theta'' \quad C \theta'' \quad r \theta'' \rightarrow s'' \quad s'' \bar{a}_k \rightarrow t''}{f \bar{e}_n \bar{a}_k \rightarrow t''}$$

and this ends the proof. \square

Although the converse of this result is not true (i.e. not all the deterministic functions are non-ambiguous), this is enough to ensure that the cuts will be safe. A more detailed characterization of semantically deterministic functions will 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.

Next we introduce two claims which establish the conditions for introducing the dynamic cut safely.

Claim 1

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

- (i) *Binding any variable in e .*
- (ii) *Computing a hnf for any expression $g(e'_1, \dots, e'_m)$ 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 have 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 $e \equiv f(e_1, \dots, e_n)$ an expression, s.t. a hnf of e is required during the computation of G . Let $f(\bar{t}) = r \Leftarrow C$ 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, \dots, e_m, e .*
- (ii) *Computing a hnf for any expression $g(e'_1, \dots, e'_m)$ 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.

<code>f X = X</code>	<code>p 0 0 = true</code>
<code>f X = X + 1</code>	<code>p 0 1 = true</code>
<code>g X = f X <== p X Y == true</code>	<code>h 1 = 1</code>

Fig. 5. Example for Claim 2

A goal like `h (g 0)` returns, without the 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 pattern 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.

incNatural	$X = X + 1$	\leq	natural	X	
natural	X	$=$	true	\leq	$X \geq 0$
zeroOrOne	$=$	0			
zeroOrOne	$=$	1			

Fig. 6. Example for Claim 2

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

In principle we could think that function `natural` could safely cut if no binding for X is produced during its condition, because regardless of whenever some non-deterministic function has been evaluated or not the function is going to produce *true* as result. But we must remember that the computation of e is a subcomputation of the initial goal, and therefore the evaluation of its condition can have some 'long distance' effects, as the following example shows.

Consider for instance the goal `incNat zeroOrOne`. The expected answers are 1 and 2 since the two values produced 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 \geq 0`. To solve the condition the non-deterministic function 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 avoided the reevaluation 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

We explain in this section how to accommodate dynamic cut into a translation scheme of the form $\mathcal{T} : \mathbf{Source} \rightarrow \mathbf{Prolog}$ for the case of the FLP language \mathcal{TOY} , but we think that it is not difficult to extend the approach to other translations schemes.

5.1 The Translation Scheme

The translation scheme for \mathcal{TOY} ¹, which can be found in [1], is the result of three stages:

- (1) The source \mathcal{TOY} program, which uses higher order syntax, is translated into \mathcal{TOY} -like programs written in first order syntax.
- (2) The compiler introduces *suspensions* [6, 11] 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 $f(e_1, \dots, e_n)$ by a Prolog term of the form $susp(f(e_1, \dots, e_n), R, S)$ (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 $f(e_1, \dots, e_n)$ is required, the variable R will be bound to the obtained value, and we say that the suspension has been evaluated. 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.
- (3) 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 $susp(f(X_1, \dots, X_n), R, S)$, a call $f(X_1, \dots, X_n, H)$ is made to a specific predicate returning in H the desired head normal form. The set $Prolog_{FS}$ consists of the clauses for those predicates, which are exactly the predicates affected by the introduction of dynamic cut.

5.2 Generating Prolog Code

Next we explain in detail the third phase (code generation), taking into account dynamic cut. This is done regarding the definitional tree of the function, and will be represented as $prolog(f, dt(f))$ where the auxiliary function *prolog/2* takes a definitional tree and a function symbol, possibly different to the function of

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

the definitional tree (this is to introduce new auxiliary functions), returning as value a set of Prolog clauses.

The interesting cases for this paper are those when code for dynamic cut can be added. For this code 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 $susp(f(e_1, \dots, e_n), R, S)$ and f is a non-deterministic function then R must be added to Vs . This is essential for performing dynamic cut safely.
 (2) If E contains an evaluated suspension $susp(f(e_1, \dots, e_n), R, S)$, 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

varlist(E, Vs), *<compute something with E>*, *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 a Prolog exercise and can be found at <http://babel.dacya.ucm.es/cut>.

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 then distinguish cases according to the shape of the root of dt .

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

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

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.

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

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

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

Assume $dt \equiv f(\bar{s}) \rightarrow \mathbf{or} \ \langle dt_1 \mid \dots \mid dt_m \rangle$

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 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 define a non-deterministic function, then code for dynamic cut cannot be added:

$$prolog(g, dt) = \{ g(\bar{s}, H) :- g_1(\bar{s}, H). \} \cup \dots \cup \{ g(\bar{s}, H) :- g_m(\bar{s}, H). \} \cup \\ prolog(g_1, dt_1) \cup \dots \cup prolog(g_m, dt_m)$$

where g_1, \dots, g_m are new function symbols.

Case 2.2 If R define a deterministic function, then we add code for dynamic cut:

$$prolog(g, dt) = \{ g(\bar{s}, H) :- \text{varlist}(\bar{s}, Vs), \\ g'(\bar{s}, H), \\ (\text{checkvarlist}(Vs), \\ ! \% \text{ this is the dynamic cut} \\ ; \\ \text{true}). \} \cup \\ \{ g'(\bar{s}, H) :- g_1(\bar{s}, H). \} \cup \dots \cup \{ g'(\bar{s}, H) :- g_m(\bar{s}, H). \} \cup \\ prolog(g_1, dt_1) \cup \dots \cup prolog(g_m, dt_m)$$

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

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 $f(\bar{s}) = e \Leftarrow C$. In this case it is always possible to add code for dynamic cut between the code for the conditions C and the code for the body e . Some care must be taken with extra variables in C , that is, variables in C not occurring in the head $f(\bar{s})$. If one of such 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 in C succeed with some bindings for existential variables, there is no need of finding alternative bindings for such variables. But if one extra variable in C occurs also in e , it might contribute to its value, and therefore to the value of $f(\bar{s})$; 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.

$$prolog(g, dt) = \{ g0(\bar{s}, H) :- \\ \text{varlist}((\bar{s}, e), Vs), \% \text{ notice the body } e \\ \text{solve}(C), (\text{checkvarlist}(Vs), \\ ! \% \text{ this is the dynamic cut} \\ ; \text{true}), \\ \text{hnf}(e, H). \}$$

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.

5.3 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/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, 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), &&'(X,Y,H), (checkvarlist(Vs), ! ; true).
.....
```

where the auxiliary predicate $\&\&'$ is defined exactly as the predicate $\&\&$ would have been defined without dynamic cut in mind.

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. We write only the code for **part**:

```
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).
```

6 Experimental results

Fig. 6 presents some experimental results obtained with the system \mathcal{TOY}^2 . Additionally 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

² Running on a PC under O.S. Linux with processor Intel Celeron at 600 MHZ and 128 Mb RAM.

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.

Program	Goal	Without Dynamic Cut	Dynamic Cut
example1.toy	G1	23.4 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

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 this case the cut cannot avoid the search of any *or* branch and the results are similar, with the code including cut slightly worse due to the run-time checking of bindings.

- 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`.

- 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 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 $G_{13} \equiv \text{false} \ \&\& \ \text{loop} == \text{true}$, where loop is defined as $\text{loop} = \text{loop}$, fails with dynamic cut while loops without it.

7 Conclusions

This work 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 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 1, can be used now without decreasing the efficiency of the computations.

Compared to a previous work ([10]) on this subject our proposal present 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.

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. In a different line, a deeper theoretical work would be desirable both extending the class of functions qualified as deterministic and providing an operational framework suitable to prove the properties of the technique.

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.
2. S. Antoy. *Definitional Trees*. Int. Conf. on Algebraic Logic Programming (ALP'92), LNCS 632, Springer Verlag 1992, 143-157.
3. S. Antoy. *Constructed-based Conditional Narrowing*. PPDP'01, ACM Press 2001, 199-206.
4. S. Antoy, R. Echahed, M. Hanus. *A Needed Narrowing Strategy*. Journal of the ACM Vol. 47, no. 4, pages 776-822, July 2000.
5. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
6. 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, pages 1-20. MIT Press, 1993.
7. R. Caballero and M. Rodríguez-Artalejo. *A Declarative Debugging System for Lazy Funcional-Logic Programs*. Electronic Notes in Computer Science. Vol. 64. Elsevier, 2002.
8. J.C. González-Moreno, T. Hortalá-González, F.J. López-Fraguas, M. Rodríguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic*. J. of Logic Programming, 40(1), pp 47-87, 1999.
9. 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>.
10. R. Loogen, St. Winkler. *Dynamic Detection of Determinism in Functional-Logic Languages*. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'91), LNCS 528, Springer Verlag 1991, 335-346.
11. R. Loogen, F.J. López-Fraguas, and M. Rodríguez-Artalejo. *A Demand Driven Computation Strategy for Lazy Narrowing*. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'93), LNCS 714, Springer Verlag 1993, 184-200.
12. F.J. López-Fraguas, and J. Sánchez-Hernández. *\mathcal{TOY} a Multiparadigm Declarative System*, In Proc. RTA'99, LNCS 1631, Springer Verlag, 244-247, 1999.