Dynamic Detection of Deterministic Computations in Non-Deterministic Functional-Logic Programs

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 [7], 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 [3, 8], 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

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. The examples are written in the concrete syntax of the lazy FLP language Toy [9] but can be easily adapted to other languages such as Curry [6]. 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 $\mathcal{TOY}$ [9] but can be easily adapted to other FLP languages like Curry [6]. 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 has a *left-hand side*, a *right-hand side* and an optional *condition*:

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

where 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, 8] 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: $e_1 == e'_1, \ldots, e_k == e'_k$, simply by assuming that an auxiliary function: $\mathsf{main}\ R_1 \ldots R_n = \mathsf{true} <== e_1 == e'_1, \ldots, e_k == e'_k$ 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 $\mathsf{main}\ R_1 \ldots R_n$. 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. We assume that goals are solved by means of an operational mechanism based on needed narrowing with sharing [3, 8], as well as a Prolog-based implementation as described in [8]. 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.

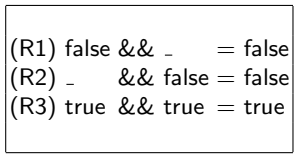

Fig. 1. The parallel and with its definitional tree

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 [7]) 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 [7], 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 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 rely 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.

```
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

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 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, one of the two types of dynamic cut is related to the existence of *or* nodes in definitional trees for semantically deterministic functions. Moreover, 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 (maybe partial) 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 [7], 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 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 possibly non deterministic 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 the following two possibilities holds:
   (a) Heads do not overlap, that is, $f(\overline{t})$ and $f(\overline{t'})$ are not unifiable.
   (b) Bodies can be fusioned, that is, 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 [4].

Claim 1 *Non-ambiguous functions are semantically deterministic functions.*

Although the converse is not true, this is enough to ensure that the cuts will be safe. This claim is a well-known result about TRS, but in our case its validity depends on a suitable definition of the operational semantics of FLP languages which is not discussed here. 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. The dynamic cut will be safe assuming the two following claims:

Claim 2 *Let G be 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 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.

Claim 3 *Let f be a function and $f(\overline{t}) = e \Leftarrow C$ is a defining rule for f used to compute a hnf of an expression $e \equiv f(e_1, \ldots, e_n)$. If the condition C is successfully computed without:*

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

Then no alternative re-evaluation of C is needed.

Although following these claims we could safely introduce dynamic cut associated to many evaluations of hnf, most of this cuts would be unnecessarily costly. 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 $\mathcal{T}$: **Source** $\rightarrow$ **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* [5, 8] 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 $f(e_1, \ldots, e_n)$ 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*,...,*X_n*),R,S)*, a call *f(*X_1*,...,*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 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

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

• *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 essential for performing dynamic cut safely.
(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 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 $dt \equiv f(\overline{s}) \to \mathbf{case}\ X\ \mathbf{of}\ \langle c_1 :\ dt_1 \ldots c_m :\ dt_m\rangle$
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 $\mathsf{g'}$ is a new function symbol.

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

Assume $dt \equiv f(\overline{s}) \to \mathbf{or}\ \langle dt_1 \mid \ldots \mid dt_m\rangle$
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 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:

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

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

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

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

where $g', g_1, \ldots, g_m$ 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 $f(\overline{s})\ =\ e \Leftarrow C$. 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 $f(\overline{s})$. 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 $f(\overline{s})$; 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.

$$
\begin{array}{l}
prolog(g, dt) = \{\ \mathsf{g0}(\overline{s},\mathsf{H})\ \mathsf{:\!-} \\
\qquad\qquad\qquad\qquad\quad \mathsf{varlist}((\overline{s}, e),\mathsf{Vs}),\ \%\ \textit{notice the body } e \\
\qquad\qquad\qquad\qquad\quad \mathsf{solve}(\mathsf{C}),\ (\mathsf{checkvarlist}(\mathsf{Vs}), \\
\qquad\qquad\qquad\qquad\quad !\ \%\ \textit{this is the dynamic cut} \\
\qquad\qquad\qquad\qquad\quad ;\ \mathsf{true}), \\
\qquad\qquad\qquad\qquad\quad \mathsf{hnf}(e,\mathsf{H}).\ \}
\end{array}
$$

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 he 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

Figure 4 presents some experimental results obtained with the system $\mathcal{TOY}$. The complete set of examples can be found at `http://babel.dacya.ucm.es/cut`. 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 includes an existential condition in a program rule that will include code for the dynamic cut.
- composed.toy: Program to check whether a number is composed, 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

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.
composed.toy	G11	151.0 sec.	0.4 sec.
composed.toy	G12	>5 hours	4.0 sec.

Fig. 4. Runtime Table

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 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.

7 Conclusions

This work presents a mechanism of *dynamic cut* for lazy FLP programs that can 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 ([7]) 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-Roldán, 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, R. Echahed, M. Hanus. *A Needed Narrowing Strategy.* Journal of the ACM Vol. 47, no. 4, pages 776-822, July 2000.
4. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
5. 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.
6. 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.
7. 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.
8. 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.
9. F.J. López-Fraguas, and Jaime Sánchez Hernández. $\mathcal{TOY}$ *a Multiparadigm Declarative System*, In Proc. RTA'99, LNCS 1631, Springer Verlag, 244-247, 1999.