Implementing Dynamic-Cut in $\mathcal{TOY}$ [1]

R. Caballero[2] Y. García-Ruiz[3]

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
Madrid, Spain

Abstract

This paper presents the integration of the optimization known as *dynamic cut* within the functional-logic system $\mathcal{TOY}$. The implementation automatically detects deterministic functions at compile time, and includes in the generated code the test for detecting at run-time the computations that can actually be pruned. The outcome is a much better performance when executing deterministic functions including either or-branches in their definitional trees or extra variables in their conditions, with no serious overhead in the rest of the computations. The paper also proves the correctness of the criterion used for detecting deterministic functions w.r.t. the semantic calculus CRWL.

Keywords: determinism, functional-logic Programming, program analysis, programming language implementation.

1 Introduction

Nondeterminism is one of the characteristic features of Logic Programming shared by Functional-Logic Programming. It allows elegant algorithm definitions, increasing the expressiveness of programs. However, this benefit has an associated drawback, namely the lack of efficiency of the computations. There are two main reasons for this:

- The complexity of the search engine required by the nondeterministic programs, which slows the execution mechanism.
- The possible occurrence of redundant subcomputations during a computation.

In the Logic Programming language Prolog, the second point is partially solved by introducing a non-declarative mechanism, the so-called *cut*. Programs using cuts are much more efficient, but at the price of becoming non-declarative.

In the case of Functional-Logic Programming the situation is somehow alleviated by the *demand driven strategy* [2,8], which is based on the use of *definitional trees*

[1] This work has been funded by the projects TIN2005-09207-C03-03 and S-0505/TIC/0407.

[2] Email: rafa@sip.ucm.es

[3] Email: ygruiz@gmail.com

[1,8]. Given any particular program function, the strategy uses the structure of the left-hand sides of its program rules in order to reduce the number of redundant subcomputations. The implementation of modern Functional-Logic languages such as $\mathcal{TOY}$ [9] or Curry [6] is based on this strategy. Our proposal also relies in the demand driven strategy, but introduces a safe and declarative optimization to further improve the efficiency of deterministic computations. This optimization is the *dynamic cut*, first proposed by Rita Loogen and Stephan Winkler in [10]. In [4,3] the same ideas were adapted to a setting including non-deterministic functions and a demand driven strategy, showing by mean of examples the efficiency of the optimization.

However, in spite of being well-known and accepted as an interesting optimization, the dynamic cut had not been implemented in any real system up to now. In this paper we present this implementation in the functional-logic system $\mathcal{TOY}$ (available at `http://toy.sourceforge.net`).

The system considers two special situations where the computations can be safely pruned: rules with existential variables in the conditions and sets of overlapping rules occurring in deterministic functions. The first situation is detected simply by examining the structure of each rule. During this process the *try* nodes of the definitional trees corresponding to program rules with existential variables in the conditions are labeled as *tryCut* nodes. The implementation of the second situation, detecting deterministic functions with overlapping rules, relies in a determinism analysis performed by the system. Using this information the system will label the *or* nodes occurring in the definitional trees of deterministic functions as *or-cut* nodes. The code generator of $\mathcal{TOY}$ has been extended to handle both *try-cut* and *orCut* nodes, including the dynamic cut code in the corresponding generated code.

The information about deterministic functions is required not only at compile time but also at run-time, when it is used for checking dynamically if the cut must take place in a particular computation. As previous works [10,4,3] have shown, this dynamic test is necessary for ensuring the correctness of the cut, i.e. that the optimization does not affect the set of solutions of any goal.

The determinism analysis performed by the system follows the well-known criterion of non-ambiguity already introduced in [10]. From the theoretical point of view, the novelty of this paper w.r.t. previous works is that we have proved formally the correction of such criterion w.r.t. the semantic calculus CRWL, proposed as suitable logic foundations for Functional-Logic Programming in [5]. Of course the completeness cannot be established because this is an undecidable property [13]. For that reason we also allow the user to select explicitly some functions as deterministic.

The paper is organized as follows. Next section introduces the non-ambiguity criterion for detecting deterministic functions and the correctness theorem. Section 3 shows by means of examples the cases where the optimization will be applied. Section 4 presents the steps followed during the implementation of the dynamic cut in $\mathcal{TOY}$, and Section 5 finalizes presenting some conclusions.

2 Detecting Deterministic Functions in Functional-Logic Programs

This section proves the correctness of the non-ambiguity condition used for detecting deterministic functions w.r.t. the semantic calculus CRWL [5].

2.1 Deterministic Functional-Logic Functions

Before defining and characterizing deterministic functions we need to establish briefly some basic notions and terminology. We refer to [5] for more detailed definitions.

We assume a *signature* $\Sigma = \langle DC, FS \rangle$, where DC and FS are ranked sets of *constructor symbols* resp. *function symbols*. Given a countably infinite set $\mathcal{V}$ of variables, we build *CTerms* (using only variables and constructors) and *Terms* (using variables, constructors and function symbols). We extend Σ with a special nullary constructor $\bot$ (0-arity constructor) obtaining a new signature $\Sigma_\bot$ and we will write $Term_\bot$ and $CTerm_\bot$ (*partial terms*) for the corresponding sets of terms in this extended signature.

A $\mathcal{TOY}$ program $\mathcal{P}$ is composed of data type declarations, type alias, infix operators, function type declarations and a set of defining rules for functions symbols. Each defining rule for a function $f \in FS$ has a *left-hand side*, a *right-hand side* and a optional *condition*: $\underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \to \underbrace{r}_{\text{right-hand side}} \Leftarrow \underbrace{C}_{\text{condition}}$

where $t_1 \ldots t_n$ must be linear Cterms and C must consist of finitely many (possibly zero) joinability statements $e_1 == e_2$ with $e_1, e_2 \in Term$. A natural *approximation ordering* $\sqsubseteq$ for partial terms can be defined as the least partial ordering over $Term_\bot$ satisfying the following properties:

- $\bot \sqsubseteq t$, for all $t \in Term_\bot$
- $X \sqsubseteq X$, for all variable X
- if $t_1 \sqsubseteq s_1, ..., t_n \sqsubseteq s_n$, then $c\ t_1 \ldots t_n \sqsubseteq c\ s_1 \ldots s_n$, for all $c \in DC^n$ and $t_i, s_i \in CTerm_\bot$.

A *partially ordered set* (*poset* in short) *with bottom* is a set S equipped with a partial order $\sqsubseteq$ and a least element $\bot$ (w.r.t. $\sqsubseteq$). $D \subseteq S$ is a *directed set* iff for all $x, y \in D$ there exists $z \in D$ such that $x \sqsubseteq z$, $y \sqsubseteq z$. A subset $A \subseteq S$ is a *cone*, iff $\bot \in A$ and for all $x \in A$, $y \in S$ $y \sqsubseteq x \Rightarrow y \in A$. An *ideal* $I \subseteq S$ is a directed cone. The program semantics is defined by the semantic calculus CRWL presented in [5]. CRWL (Constructor Based ReWriting Logic) is a theoretical framework for the lazy functional logic programming paradigm. Given any program P, CRWL proves statements of the form $e \to t$ with $e \in Term_\bot$ and $t \in CTerm_\bot$. We denote by $\mathcal{P} \vdash_{\text{CRWL}} e \to t$ that the statement $e \to t$ can be proved in CRWL w.r.t. P. The intuitive idea is that t is a valid approximation of e in $\mathcal{P}$. The *denotation* of any $e \in Term_\bot$, written $[\![e]\!]$, is defined as: $[\![e]\!] = \{t \in CTerm_\bot \mid P \vdash_{\text{CRWL}} e \to t\}$.

Now we are ready for presenting the formal definition of deterministic function in our setting.

Definition 2.1 (**Deterministic Functions**)
Let f be a function defined in a program $\mathcal{P}$. We say that f is a deterministic function iff $[\![f\,\bar{t}_n]\!]$ is an ideal for every $\bar{t}_n$ s.t. t_i is a $\text{CTerm}_\perp$ for all $i = 1 \ldots n$.

We call *non-deterministic* to the functions that don't fulfill the previous definition. The intuitive idea behind a deterministic function is that given some arbitrary ground values it can return at most one result [7]. However, in a lazy setting whenever a function can return a value t it is expectable that it can return all the less defined terms s, $s \sqsubseteq t$ as well. The previous definition takes this idea into account. Consider for instance the following small program:

$$\mathsf{data\ pair = pair\ int\ int} \qquad \mathsf{f\ 1 = pair\ 1\ 2} \qquad \mathsf{g\ 1 = 1} \qquad \mathsf{g\ 1 = 2}$$

Using CRWL it can be proved that $[\![f\ 1]\!] = \{\perp, \text{pair} \perp \perp, \text{pair } 1\ \perp, \text{pair} \perp\ 2, \text{pair } 1\ 2\}$, $[\![f\ t]\!] = \{\perp\}$ if $t \neq 1$, $[\![g\ 1]\!] = \{\perp, 1, 2\}$, $[\![g\ t]\!] = \{\perp\}$ if $t \neq 1$. Then g is a non-deterministic function because for the parameter 1 the set $\{\perp, 1, 2\}$ is not an ideal, in particular because it is not directed: taking $x = 1, y = 2$ is not possible to find $z \in \{\perp, 1, 2\}$ s.t. $x \sqsubseteq z$, $z \sqsubseteq 2$. On the contrary, it is easy to check that f is a deterministic function.

2.2 Non-ambiguous functions

The definition 2.1 is only a formal definition and cannot be used in practice. In [4] an adaptation of the non-ambiguity condition of [11] is presented, which we will use as an easy mechanism for the effective recognition of deterministic functions. Not all the deterministic functions are non-ambiguous. However, the non-ambiguity criterion will be enough for detecting several interesting deterministic functions.

Definition 2.2 (Non-ambiguous functions)
Let $\mathcal{P}$ 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 $f\ \bar{t}_n = e \Leftarrow C$ is a defining rule for f, then $var(e) \subseteq var(\bar{t})$ and all function symbols in e belong to F.

(ii) For any pair of variants of defining rules for f, $f\ \bar{t}_n = e \Leftarrow C$, $f\ \bar{t'}_n = e' \Leftarrow C'$, one of the following two possibilities holds:
   (a) Left-hand sides do not overlap, that is, the terms $(f\ \bar{t}_n)$ and $(f\ \bar{t'}_n)$ are not unifiable.
   (b) If θ is the m.g.u. of $f\ \bar{t}_n$ and $f\ \bar{t'}_n$, then $e\theta \equiv e'\theta$.

In [3,4] the inclusion of the set on non-ambiguous functions in the set of deterministic functions was claimed. Here, and thank to the previous formal definition, we can prove this result:

Theorem 2.3 . *Let $\mathcal{P}$ be a program and f be a non-ambiguous function defined in $\mathcal{P}$. Then f is deterministic.*

Proof. (see Appendix A, which is intended only for the referees and not for the final version) $\square$

The non-ambiguity condition characterizes set of functions F as deterministic.

```
% P1: 'Parallel' multiplication
data nat            = zero | s nat

add zero Y          = Y
add (s X) Y         = s (add X Y)
-----------------------------------------------
multi zero _        = zero
multi _ zero        = zero
multi (s X) (s Y)   = s (add X (add Y (multi X Y) ))
-----------------------------------------------
power N zero        = s zero
power N (s M)       = multi N (power N M)

odd zero            = false
odd (s zero)        = true
odd (s (s N))       = odd N

toNat N             = if (N==0) then zero
                        else s (toNat (N-1))
```

```
% P2: 'Classical' multiplication
data nat            = zero | s nat

add zero Y          = Y
add (s X) Y         = s (add X Y)
-----------------------------------------------
multi zero _        = zero
multi (s X) Y       = add Y (add X Y)
-----------------------------------------------
power N zero        = s zero
power N (s M)       = multi N (power N M)

odd zero            = false
odd (s zero)        = true
odd (s (s N))       = odd N

toNat N             = if (N==0) then zero
                        else s (toNat (N-1))
```

Fig. 1. Two methods for multiplying

This is because the value of a function may depend on other functions, and in general this dependence can be mutual. In practice the implementation starts with an empty set F of non-ambiguous functions, adding at each step to F those functions that satisfy the definition and that only depend on functions already in F. This is done until a fix value for F is reached.

Although most of the deterministic functions that occur in a program are non-ambiguous as well, there are some functions which are not detected. This happens for instance in the function f of following example: f 1 = 1 f 1 = g 1 g 1 = 1. It would be useful to use additional determinism criteria, such as those based on abstract interpretation proposed in [12], but the detection of deterministic function will be still incomplete. For that reason the system allows programmer to distinguish deterministic functions annotating them by using --> instead of =, as in the following example: f 1 --> 1 f 1 --> g 1 g 1 = 1,
which indicates that f is deterministic. The non-annotated functions like g will be analyzed following the non-ambiguity criterion.

3 Pruning Deterministic Computations

In this section we present briefly the two different situations where the dynamic cut can be introduced.

3.1 Deterministic Functions Defined through Overlapping Program Rules

Sometimes deterministic functions can be defined in a natural way by using overlapping rules. Consider for instance the two programs of Figure 1. Both programs contain functions for computing arithmetic using Peano's representation. The function toNat is used for easily converting positive numbers of type int to their Peano representation. The only difference between P_1 and P_2 is the method for multiplying numbers. The function multi at P_2, which we have called 'classical' reduces the first argument before each recursive call until it becomes zero. The method multi

X	Y	P_1	P_2
0	100000	0	0
0	50000	0	0
100	1000	2.7	2.7
400	400	4.1	4.1
1000	100	4.9	_
50000	0	0	3.5
100000	0	0	_

multi (toNat X) (toNat Y)

N	P_1	P_2
10^4	0.7	0
10^5	6.1	0
10^6	60.0	0
10^7	_	0

odd (power zero (toInt N))
without dynamic cut

N	P_1	P_2
10^4	0	0
10^5	0	0
10^6	0	0
10^7	0	0

odd (power zero (toInt N))
with dynamic cut

Fig. 2. Comparative tables

at P_1, which we have called 'parallel', reduces both arguments before the recursive call. Observe that the first two rules of multi in P_1 are overlapping. However it is easy to check that it is a non-ambiguous and hence a deterministic function.

The first table at Figure 2 shows the time[4] required for solving goals of the form multi (toNat X) (toNat Y) == R in both programs. The symbol _ means that the system has run out of memory for the goal. From these data is clear that the parallel multi of P_1 behaves better than its classical counterpart of P_2. The reason is that in P_1 the computation of multi reduces the two arguments simultaneously saving both time and space. However this kind of 'parallel' definitions are not used very often in Functional-Logic Programming because programmers know that overlapping rules can produce unexpected behaviors due to the backtracking mechanism. Indeed using P_1 a goal like multi zero zero == R has two solutions, both giving to R the value zero, instead as only one as expected (and as the program P_2 does). Such redundant computation can affect the efficiency of other computations. The central table of Figure 2 contains the time required by both programs for checking if the N-th power of zero is odd without the dynamic cut optimization. The goal returns **no** in both cases as expected, but we observe that now P_1 behaves rather worse than P_2, even running out of memory for large enough numbers. This is because the subgoal power zero (toInt N) needs to compute N multiplications, and in P_1 this means N redundant computations. Thus using P_1 without dynamic cut the goal odd (power zero (toInt N)) will check N times if zero is odd, while in P_2 this is done only once. The dynamic cut solves this situation, detecting that multi in P_1 is a deterministic function and cutting the possibility of using the second rule of multi if the first one has succeeded producing a result (and satisfying some conditions explained below). The third table, at the right of Figure 2 has been obtained after activating the dynamic cut. The problem of the redundant computations has been solved. It is worth pointing out that the times of the first table do not change after activating the optimization.

3.2 Existential variables in conditions

Consider now the program of Figure 3. It includes a simple representation of DNA molecules, which are build by two chains of nucleotides. The nucleotides of the two strands are connected in compatible pairs, defined in the program through function compatible. The function dna detects if its two input parameters represent two

[4] All the results displayed in seconds, obtained on a computer at 2.13 GHz with 1 Gb of RAM

```
data nucleotides = adenine | guanine | cytosine | thymine

compatible adenine thymine = true
compatible thymine adenine = true
compatible guanine cytosine = true
compatible cytosine guanine = true

dna [] [] = true
dna [N1|R1] [N2|R2] = true <== compatible N1 N2, (dna R1 R2)

dnaPart S1 S2 L = true <== part P1 S1 L , part P2 S2 L, dna P1 P2

part X Y L = true <== (U ++ X) ++ V == Y, length X == L
```

Fig. 3. Detecting DNA strands

strands that can be combined in a DNA molecule. Function dnaPart checks if the two input sequences S1 and S2 contain some subsequences P1 and P2 of length L that can occur associated in a DNA molecule. This function relies in function part which checks if the parameter X is sublist of length L of the list Y. The functions ++ and length, represent respectively the concatenation of lists and the number of elements in a list. Consider the following session in the system $\mathcal{TOY}$:

```
Toy> dnaPart (repeat 1000 adenine) (repeat 1000 thymine) 5
    yes.    Elapsed time: 844 ms.
    more solutions? y
    yes.    Elapsed time: 40390 ms.
```

The goal dnaPart (repeat 1000 adenine) (repeat 1000 thymine) 5 asks if in two strands of 1000 nucleotides of adenine and thymine respectively it is possible to find two subsequences of 5 nucleotides, one from each strand, which can occur associated in a DNA molecule. The answer given by the system after 0.8 seconds is yes (actually *all* the subsequences of n elements of the first strand are compatible with all the subsequences of n elements of the second strand). If the user asks for a second answer, the same redundant answer yes is obtained after more than 40 seconds. The second answer is useless because it doesn't provide new information, and greatly affects the efficiency. It can be argued that there is no point in asking for a second answer after the first, but this situation can occur as subcomputations of a bigger computation and cannot be avoided in general.

Examining the code we find out easily the source of the redundant computation: the condition of function part includes two existential variables U and V. When the user asks for more solutions the backtracking mechanism looks for new values of the variables satisfying the conditions. But this is unnecessary because the rule already as returned true and cannot return any new value. The dynamic cut will avoid this redundant computation. Here is the same goal running after activating the dynamic cut optimization in $\mathcal{TOY}$:

```
Toy>dnaPart (repeat 1000 adenine) (repeat 1000 thymine) 5
    yes.      Elapsed time: 844 ms.
    more solutions ? y
    no.      Elapsed time: 0 ms.
```

Now the system detects automatically that there are no more possible solutions after the first one, reducing the 40 seconds to 0. The interested reader can find in [4] more experimental results. The experiments in that paper were tested introducing manually the code for the dynamic cut before the optimization was part of the system. However the results have been confirmed by the current implementation.

3.3 Dynamic conditions for the cut

From the previous examples one could consider that the cut can be introduced safely in the code of functions multi and part without taking into account any run-time test. But the cut also depends on dynamic conditions. There are two situations that must be taken into account before applying the cut:

i) Variable bindings.

Consider the goal: multi X zero == R, with X a logical variable. Using the program P_1 of Figure 1 this goal produces two answers: { X↦zero, R↦zero } and { R↦zero }. The first answer is obtained using the first rule for multi and the second answer through the second rule. Introducing a cut after the first answer would be unsafe; the second answer is not redundant, but gives new information w.r.t. the first one. As it includes no binding for X it can be interpreted as *'for every X, the equality* multi X zero == zero *holds'*, and therefore subsumes the first answer.

ii) Non deterministic functions computed.

Suppose we include a new function zeroAndOne in the program P_1 of Figure 1 defined as: zeroAndOne = zero zeroAndOne = s zero

Then a goal like multi zeroAndOne (s zero) == R will return two answers: { R ↦ zero } and { R ↦ s zero }. Introducing the cut after the first answer would be again unsafe. But in this case it is not because it prevents the use of the second rule, but because it would avoid the backtracking of the non-deterministic function zeroAndOne that leads to the application of the third rule of multi, yielding the second answer.

Therefore the cut must not take place if after obtaining the first result of the deterministic function any of the variables in the input arguments has been bound or a non-deterministic function has been computed. As we will see in the following paragraph the implementation generates a dynamic test for checking these conditions before introducing the cut.

4 Implementing the Dynamic Cut

4.1 Compiling programs into Prolog

The $\mathcal{TOY}$ compiler transforms $\mathcal{TOY}$ programs into Prolog programs following ideas described in [8]. A main component of the operational mechanism is the computation of *head normal forms* (hnf) for expressions. The translation scheme can be divided into three phases:

1) Higher order $\mathcal{TOY}$ programs are translated into programs in first order syntax.

2) Function calls $f(e_1, \ldots, e_n)$ occurring in the first order $\mathcal{TOY}$ program rules are replaced by Prolog terms of the form $susp(f(e_1, \ldots, e_n), R, S)$ called *suspensions*. The logical variable S is a flag which it is bound to a concrete value, say hnf, once the suspension is evaluated. R contains the result of evaluating the function call.

Its value is meaningful only if S==hnf holds.

3) Finally the Prolog clauses are generated, adding code for *strict equality* and *hnf* (to compute head normal forms). Each n-ary function f is translated into a Prolog predicate $f(X_1,\ldots,X_n,H)$. When computing a hnf for an unevaluated suspension *susp(f(*X_1*,...,*X_n*),R,S)*, a call *f(*X_1*,...,*X_n*,H)* will occur in order to obtain in H the desired head normal form.

We are particularly interested in the third phase (code generation), since it will be affected by the introduction of dynamic cuts. Before looking more closely at this phase we need to introduce briefly our notation for definitional trees.

4.2 Definitional Trees in $\mathcal{TOY}$

Before generating the code for any function the compiler builds its associated definitional tree. In our setting the definitional tree dt of a function f, can be of either of the following three forms:

- $dt(f) = f(\bar{t}_n) \rightarrow case\ X\ of\ \langle c_1(\overline{X}_{m_1}) : dt_1; \ldots ; c_k(\overline{X}_{m_k}) : dt_k\rangle$, where X is the variable at position u in $f(\bar{t}_n)$ and $c_1 \ldots c_k$ are constructor symbols, with dt_i a definitional tree for $i = 1 \ldots k$.
- $dt(f) = f(\bar{t}_n) \rightarrow or\ \langle dt_1 \mid \ldots \mid dt_k\rangle$, with dt_i a definitional tree for $i = 1 \ldots k$.
- $dt(f) = f(\bar{t}_n) \rightarrow try\ (r \Leftarrow C)$, with $f\ \bar{t}_n = r \Leftarrow C$ corresponding to an instance of a program rule for f.

If each case we say that the tree has a case/or/try node at the root, respectively. A more precise definition together with the algorithm that produces a definitional tree from a function definition can be found in [8]. The only difference is that we do not allow 'multiple tries', i.e. *try* nodes including 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. As an example of definitional tree, consider again the definition of function multi in the program P_1 of Figure 1:

```
(R1)   multi zero _          = zero
(R2)   multi _ zero          = zero
(R3)   multi (s X) (s Y)     = s (add X (add Y (multi X Y) ))
```

Its definitional tree, denoted as *dt(multi)*, is defined in $\mathcal{TOY}$ as:

```
dt(multi) =
  multi(A,B)→ or ⟨
     multi(A,B)→ case A of
          ⟨ zero    : multi (zero, B) → try (zero) % (R1)
          ; s(X)    : multi (s(X),B) → case B of
                         ⟨ s(Y) : multi (s(X), s(Y)) → try (s (add C (add D (multi(C,D))))) % R3
          ⟩
     | multi(A,B)→ case B of ⟨ zero: multi (A,zero) → try (zero) % R2
  ⟩
```

4.3 Definitional trees with cut

From the definitional tree dt of each function the system $\mathcal{TOY}$ generates a definitional tree with cut, *dtc*. Definitional trees with cut have the same structure as usual definitional trees. The only difference is that they rename some or and try nodes as orCut and tryCut, respectively. We define a function Γ transforming a definitional tree dt into its corresponding definitional tree with cut straightforwardly by distinguishing cases depending on the root node of dt:

- $\Gamma(\ f(\bar{t}_n) \rightarrow case\ X\ of\ \langle c_1(X_{m_1}) : dt_1; \ldots ; c_k(X_{m_k}) : dt_k\rangle\ ) =$
  $f(\bar{t}_n) \rightarrow case\ X\ of\ \langle c_1(X_{m_1}) : \Gamma(dt_1); \ldots ; c_k(X_{m_k}) : \Gamma(dt_k)\rangle$
- $\Gamma(\ f(\bar{t}_n) \rightarrow or\langle dt_1 \mid \ldots \mid dt_k\rangle\ ) =$
  $f(\bar{t}_n) \rightarrow orCut\ \langle \Gamma(dt_1) \mid \ldots \mid \Gamma(dt_k)\rangle$, if f is deterministic.
- $\Gamma(\ f(\bar{t}_n) \rightarrow or\langle dt_1 \mid \ldots \mid dt_k\rangle\ ) =$
  $f(\bar{t}_n) \rightarrow or\ \langle \Gamma(dt_1) \mid \ldots \mid \Gamma(dt_k)\rangle$, if f is non-deterministic.
- $\Gamma\ (f(\bar{t}_n) \rightarrow try\ (r \Leftarrow C) = f(\bar{t}_n) \rightarrow tryCut\ (r \Leftarrow C)$ if some existential variable occurs in C (i.e. some variable occurs in C but not in the rest of program rule).
- $\Gamma\ (f(\bar{t}_n) \rightarrow try\ (r \Leftarrow C) = f(\bar{t}_n) \rightarrow tryCut\ (r \Leftarrow C)$ if no existential variable occurs in C.

For instance the dt of function multi displayed above is transformed into the following definitional tree with cut *dct* (denoted *dtc(multi)*):

```
dt(multi) =
  multi(A,B)→ orCut ⟨
     multi(A,B)→ case A of
          ⟨ zero    : multi (zero,B) → try (zero) % (R1)
          ;  s(X)   : multi (s(X),B) → case B of
                         ⟨ s(Y) : multi (s(X),s(Y)) → try (s (add C (add D (multi(C,D))))) % R3
          ⟩
     | multi(A,B)→ case B of ⟨ zero: multi (A,zero) → try (zero) % R2
  ⟩
```

Notice that the only difference corresponds to the root, which has been transformed into a orCut node because multi is a deterministic function.

4.4 Generating the code

Now we can describe the function $prolog(f, dtc)$ which generates the code for a function f from its definitional tree with cut *dtc*. The function definition depends on the node found at the root of *dtc*. There are five possibilities:

Case 1. $dtc = f(\bar{s}) \rightarrow$ **case** X **of** $\langle c_1(X_{m_1}) : dtc_1; \ldots ; c_m(X_{m_k}) : dtc_m\rangle$. Then:

$$prolog(g, dtc) = \{g(\bar{s}, H) :-\ hnf(X, HX),\ g'(\bar{s}\sigma, H).\} \cup$$
$$prolog(g', dtc_1) \cup \ldots \cup prolog(g', dtc_m)$$

where $\sigma = X/HX$ and g' is a new function symbol. The first call to hnf ensures that the position indicated by X is already in head normal form, and that therefore can be used in order to distinguish the different alternatives.

Case 2. $dtc = f(\bar{s}) \rightarrow$ **or**$\langle dtc_1 \mid \ldots \mid dtc_m\rangle$. Then:

$$prolog(g, dtc) = \{g(\bar{s}, H) :- \ g_1(\bar{s}, H).\} \ \cup \ldots \cup \ \{g(\bar{s}, H) :- \ g_m(\bar{s}, H).\} \cup$$
$$prolog(g_1, dtc_1) \cup \ldots \cup prolog(g_m, dtc_m)$$

where $g_1, \ldots, g_m$ are new function symbols. In this case each new function symbol represent one of the non-deterministic choices.

Case 3. $dtc = f(\bar{s}) \rightarrow \mathbf{orCut}\langle dtc_1 \mid \ldots \mid dtc_m\rangle$. Then

$$\begin{aligned} prolog(g, dtc) = \quad & \{g(\bar{s}, H) :- \ varlist(\bar{s}, V_s), \ g'(\bar{s}, H), \\ & \qquad (checkvarlist(V_s), \ !, \ ; \ true). \} \cup \\ & \{g'(\bar{s}, H) :- \{g_1(\bar{s}, H).\} \ \cup \ldots \ \cup \ \{g'(\bar{s}, H) :- \ g_m(\bar{s}, H).\} \cup \\ & prolog(g_1, dtc_1) \cup \ldots \cup prolog(g_m, dtc_m) \end{aligned}$$

where g', $g_1, \ldots, g_m$ are new function symbols. Observe the differences with the case 2:

- A new function g' is used as an intermediate auxiliary function between g and the non-deterministic choices.
- g starts calling a predicate varlist. This predicate, whose definition is tedious but straightforward returns in its second parameter V_s a list containing all the logical variables in the input parameters, including those used as flags for detecting the evaluation of suspensions of non-deterministic functions.
- After g' succeeds, i.e. after an or-branch has produced a result, the test for the dynamic cut is performed. This test, represented by predicate checkvarlist, checks if any of the variables in the list produced by varlist has been bound. This will mean that either an input logical variable has been bound or a non-deterministic function has been evaluated. In any of this cases the cut is avoided. Otherwise the dynamic cut, which is implemented as an ordinary Prolog cut, is safely performed. The definition of checkvarlist is simple:

  ```
  checkVarList([]).

  checkVarList([X|Xs]):- var(X), \+varInList(X,Xs), checkVarList(Xs).
  ```

  The literal \+varInList(X,Xs), checks if the variable X occurs twice in the list, detecting bindings among variables of the list.

Case 4. $dtc = \mathbf{try}\ (e \Leftarrow l_1 == r_1, \ldots, l_n == r_n)$. Then

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

If all the equalities in the conditions are satisfied the program rule returns the head normal form of its right-hand side e.

Case 5. $dtc = \mathbf{tryCut}\ (e \Leftarrow l_1 == r_1, \ldots, l_n == r_n)$. Then

$$prolog(g, dtc) = \{g(\bar{s}, H) :- varlist((\bar{s}, e), V_s),$$
$$equal(l_1, r_1), \ldots,\ equal(l_n, r_n),$$
$$(checkvarlist(V_s), !;\ \ true),$$
$$hnf(e, H).\}$$

This case is similar to the case of the orCut. The main difference is that in this case we also collect the possible new variables of the right-hand side, because if the condition binds any of them the cut must be discarded.

4.5 Examples

New we show the Prolog code generated by $\mathcal{TOY}$ for some of the function examples presented through the paper:

- Prolog code for function part of Figure 3:

```
part(A, B, C, true):-
        varList( [A, B, C ], Vs ),
        equal(susp( ++, [ susp(++, [D,A]),J]),B),
        equal(susp(length, [A]), C),
        (checkVarList(Vs), !;  true).
```

  This corresponds to the implementation of a *tryCut* node. The variables of the right-hand side are not included because in this rules was the ground term true.

- Prolog code for function multi of Figure 1

```
multi(A, B, H):-
    varList([A,B], Vs),
    multi'(A, B, H),
    (checkVarList(Vs), ! ; true ).

multi'(A, B, H):-
    hnf(A, F),
    multi'_1(F, B, H).
multi'(A, B, zero):-
    hnf(B, zero).

multi'_1(zero, B, zero).
multi'_1(s(X), B, s(susp(add,[X,susp(add,[Y,susp(multi,[X,Y])])]))):-
    hnf(B, s(Y)).
```

  The code of this example corresponds to the implementation of an *orCut* node. The two branches are represented here be the two clauses for $multi'$ (corresponding to function g' in the case 3 of the previous subsection). The cut is introduced if the first alternative, which corresponds to a *case* node with two possibilities, succeeds.

5 Conclusions

In this paper we have presented the implementation of the dynamic cut optimization in the Functional-Logic system $\mathcal{TOY}$. The optimization improves dramatically the efficiency of the computations in the situations explained in the paper. Moreover, we claim that in practice it allows the use of some elegant and expressive function definitions that were disregarded due to its inefficiency up to now.

The cut is introduced automatically by the system following the next steps:

(i) The deterministic functions of the program are detected using the non-ambiguity criterium. The correctness of the criterium in ensured by theorem 2.3. Also the user can indicate explicitly that any function is deterministic.

(ii) The definitional tree associated to each program function is examined. The *or* nodes occurring in deterministic functions are labeled during this process as *or-cut* nodes. Also the *try* nodes corresponding to program rules including existential variables in the conditions are labeled as *try-cut* nodes.

(iii) During the code generation the system will generate the dynamic cut code for *or-cut* and *try-cut* nodes. However the cut only will be performed is the dynamic conditions explained in subsection iii are fulfilled.

We think that a similar scheme might also be used for incorporating the dynamic cut to the Prolog-based implementations of the Curry language [6].

Currently the dynamic cut must be turn on in $\mathcal{TOY}$ by typing the command /`cut` at the prompt. However, we have checked that the optimization produces almost no overhead in the cases where it cannot be applied, and we plan to provide it activated by default in the future versions of the system.

References

[1] Antoy, S., *Definitional trees*, in: *Int. Conf. on Algebraic Logic Programming (ALP'92)*, number 632 in LNCS (1992), pp. 143–157.

[2] Antoy, S., R.Echahed and M. Hanus, *A needed narrowing strategy*, Journal of the ACM **47** (2000), pp. 776–822.

[3] Caballero, R. and F. López-Fraguas, *Dynamic-cut with definitional trees*, in: *Proceedings of the 6th International Symposium on Functional and Logic Programming, FLOPS 2002*, number 2441 in LNCS (2002), pp. 245–258.

[4] Caballero, R. and F. Lpez-Fraguas, *Improving deterministic computations in lazy functional logic languages*, Journal of Functional and Logic Programming **2003** (2003).

[5] Gonzlez-Moreno, J., M. Hortal-Gonzlez, F. Lpez-Fraguas and M. Rodrguez-Artalejo, *An approach to declarative programming based on a rewriting logic*, The Journal of Logic Programming **40** (1999), pp. 47–87.

[6] Hanus, M., *Curry: An Integrated Functional Logic Language (version 0.8.2. march 28, 2006)*, Available at: http://www.informatik.uni-kiel.de/ curry/papers/report.pdf (2006).

[7] Henderson, F., Z. Somogyi and T. Conway, *Determinism analysis in the mercury compiler* (1996). URL `citeseer.ist.psu.edu/henderson96determinism.html`

[8] Loogen, R., F. López-Fraguas and M. Rodríguez-Artalejo, *A demand driven computation strategy for lazy narrowing*, in: *Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'93)*, number 714 in LNCS (1993), pp. 184–200.

[9] Loogen, R., F. López-Fraguas and M. Rodríguez-Artalejo, *Toy: a multiparadigm declarative system*, in: *Int. Symp. RTA'99*, number 1631 in LNCS (1999), pp. 244–247.

[10] Loogen, R. and S. Winkler, *Dynamic detection of determinism in functional-logic languages*, in: *Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'91)*, number 528 in LNCS (1991), pp. 335–346.

[11] Loogen, R. and S. Winkler, *Dynamic detection of determinism in functional logic languages*, in: J. Maluszynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming: Proc. of the 3rd International Symposium PLILP'91, Passau*, Springer, Berlin, Heidelberg, 1991 pp. 335–346.

[12] na, R. P. and C. Segura, *Non-determinism analyses in a parallel-functional language*, Journal of Logic Programming **2004** (2005), pp. 67–100.

[13] Sawamura, H. and T. Takeshima, *Recursive Unsolvability of Determinacy, Solvable Cases of Determinacy and Their Applications to Prolog Optimization*, in: *Proceedings of the Symposium on Logic Programming*, 1985, pp. 200–207.

Appendix A

In this appendix we present the proof of theorem 2.3. Since it uses widely the CRWL calculus we first introduce the calculus inference rules:

5.1 The CRWL calculus

CRWL is an inference system consisting of six inference rules:

BT Bottom: $\dfrac{}{e \to \bot}$ **RF** Reflexivity: $\dfrac{}{X \to X}$

DC Decomposition $\dfrac{e_1 \to t_1 \ \ldots \ e_m \to t_m}{c\, e_1 \ \ldots \ e_m \to c\, t_1 \ \ldots \ t_m}$ $c \in CD^n \cup FS^{n+1},\ m \leq n,\ t_i \in CTerm_\bot$

FA Function Application: $\dfrac{e_1 \to t_1 \ \ldots, e_n \to t_n \quad C \quad r \to a \quad a\ a_1 \ \ldots \ a_k \to t}{f\, e_1 \ \ldots \ e_n\, a_1 \ \ldots \ a_k \to t}$ $(k \geq 0)$

if $t \neq \bot, \quad (f\, t_1 \ \ldots \ t_n \to r \Leftarrow C) \in [R]_\bot$

JN Join: $\dfrac{e_1 \to t \quad e_2 \to t}{e_1 == e_2}$ $t \in CTerm$

The notation $[\mathcal{R}]_\bot$ In rule **FA** represents the set of all the possible instances of program rules, where each particular instance is obtained from some function defining rule in $\mathcal{R}$, by some substitution of (possibly partial) terms in place of variables. See [5] for a detailed description of this and related calculi.

5.2 Previous Lemmata

The first two lemmata establish substitution properties that will play an important role in the proof. The lemmata use the symbol $CSubst$ for the set of all the *c-substitutions*, which are mappings $\theta : \mathcal{V} \to CTerm$, and the notation $CSubst_\bot$ for the set of all the *partial c-substitutions* $\theta : \mathcal{V} \to CTerm_\bot$ defined analogously. We note as $t\theta$ the result of applying the substitution θ to the term t.

Lemma 5.1

Let $t \in CTerm, s \in CTerm_\bot$ be such that $t \sqsubseteq s$. There there exists a substitution $\theta \in CSubst_\bot$ verifying $t\theta = s$.

Proof. Straightforward by using induction on the structure of t. □

Lemma 5.2

Let $t, t' \in CTerm$ be such that: 1) t, t' are linear, 2) $var(t) \cap var(t') = \emptyset$ and 3) There exists γ = m.g.u.(t,t'). Let $s \in Cterm_\bot$ be a term and $\theta, \theta' \in CSubst_\bot$ such that $t\theta \sqsubseteq s$, $t'\theta' \sqsubseteq s$. Then there exists a substitution θ'' s.t. $t\gamma\theta'' = t'\gamma\theta'' = s$.

Proof.
We define $u = t\gamma = t'\gamma$. u cannot be $\bot$ because this would mean that t and t' are variables X and Y resp. and that $\gamma = \{X \mapsto \bot, Y \mapsto \bot\}$, but in this case γ would not be the m.g.u. of t and t'. Observe also that it is easy to check that u must be linear because t and t' are two linear terms with no common variables and γ is their most general unifier.

We next prove that exists a substitution θ'' s.t. $u\theta'' = s$ by using induction on the structure of the CTerm u.

- If $u = X$, we define θ'' as $\{X \mapsto s\}$. Obviously $u\theta'' = s$.
- If $u = c$, $c \in DC^0$ then it can be that: 1) $t = c$ and t' a variable X, or 2) $t' = c$ and t a variable X, or 3) $t = t' = c$. In any case s must be c and the result holds defining $\theta'' = \{X \mapsto c\}$ in cases 1), 2) and $\theta'' = id$ in the case 3).
- If $u = c(u_1, \ldots, u_n)$, we distinguish again three possibilities:
  · If $t = c(u_1, \ldots, u_n)$ and $t' = X$, then s must be of the form $s = c(s_1, \ldots, s_n)$ with $u_i\theta \sqsubseteq s_i$ for $i = 1, \ldots, n$. Since u es linear, using the induction hypotheses there exist substitutions θ''_i s.t. $u_i\theta''_i = s_i$. Then we can define the substitution θ'' as:

$$\theta''(X) = \begin{cases} \theta''_i(X) & \text{if } X \in var(u_i) \text{ for some } i,\ 1 \le i \le n \\ X & \text{otherwise} \end{cases} \quad \text{and } u\theta'' = s \text{ holds.}$$

  · If $t = X$ y $t' = c(u_1, \ldots, u_n)$ is analogous to the previous case.
  · If $t = c(t_1, \ldots, t_n)$ and $t' = c(t'_1, \ldots, t'_n)$, since t and t' are linear and unifiable, there must exist substitutions $\gamma_i = u.m.g(t_i, t'_i)$ for $i = 1 \ldots n$ such that $t_i\gamma_i = t'_i\gamma_i = u_i$. s must then be of the form $s = c(s_1, \ldots, s_n)$ with $t_i\theta \sqsubseteq s_i$, $t'_i\theta' \sqsubseteq s_i$.
    Then, by the induction hypothesis there exist substitutions θ''_i with domain the variables of u_i such that $u_i\theta''_i = s_i$. We define θ'' as:

$$\theta''(X) = \begin{cases} \theta''_i(X) & \text{if } X \in var(u_i) \text{ for some } i,\ 1 \le i \le n \\ X & \text{otherwise} \end{cases} \quad \text{and } u\theta'' = s \text{ holds.}$$

□

The next lemma introduces some basic properties of CRWL . They can be proved by straightforward induction on the depth of the CRWL proof and are omitted.

Lemma 5.3
Let $\mathcal{P}$,$e \in Term_\perp$ be a program and a partial term respectively. Then:

i) Let $t, t' \in CTerm_\perp$ be such that $\mathcal{P} \vdash_{CRWL} e \to t$ and $t' \sqsubseteq t$. Then $\mathcal{P} \vdash_{CRWL} e \to t'$.

ii) Let $\mathcal{P}$ be a program and $e \in Term_\perp$ and $\theta \in CSubst_\perp$ be s.t. $\mathcal{P} \vdash_{CRWL} e\theta \to t$. Then $\mathcal{P} \vdash_{CRWL} e\theta' \to t$ for all θ' s.t. $\theta \sqsubseteq \theta'$.

iii) Let $\bar{e}_n$ s.t. $e_i \in Term_\perp$ for all $i = 1 \ldots n$, and s.t. $\mathcal{P} \vdash_{CRWL} e\ \bar{e}_n \to t$, and $a \in Term_\perp$ such that $e \sqsubseteq a$. Then $\mathcal{P} \vdash_{CRWL} a\ \bar{e}_n \to t$.

5.3 Theorem proof

Now we can prove that every non-ambiguous function is a deterministic function, as theorem 2.3 states:

Proof. In order to check that f is a deterministic function, we must prove that $[\![f\ \bar{t}_n]\!]$ is an ideal, for every $\bar{t}_n$ with $t_i \in CTerm_\perp$ for $i = 1 \ldots n$. Thus, we must prove that $[\![f\ \bar{t}_n]\!]$ is both a cone an a directed set.

- $[\![f\ \bar{t}_n]\!]$ is a cone.

- $\perp$ is in $[\![f\ \bar{t}_n]\!]$ by the inference rule **BT**.
- Let $t \in [\![f\ \bar{t}_n]\!]$ be s.t. $t \in CTerm_\perp$, i.e. $\mathcal{P} \vdash_{\text{CRWL}} f\ \bar{t}_n \rightarrow t$. Let $t' \in CTerm_\perp$ be s.t. $t' \sqsubseteq t$. Then by lemma 5.3.i), $\mathcal{P} \vdash_{\text{CRWL}}\ f\ \bar{t}_n \rightarrow t'$ and hence $t' \in [\![f\ \bar{t}_n]\!]$.

- $[\![f\ \bar{t}_n]\!]$ is a directed set.

We prove a more general result: *Consider $e \in Term_\perp$ and suppose that all the function symbols occurring in e are correspond to non-ambiguous functions. Then, $[\![e]\!]$ is a directed set.*

Let t, t' be s.t. $t, t' \in Cterm_\perp$ verifying

$$(R1) : \mathcal{P} \vdash_{\text{CRWL}} e \rightarrow t \qquad (R2) : \mathcal{P} \vdash_{\text{CRWL}} e \rightarrow t'$$

We next prove that exists some $s \in Cterm_\perp$ such that: a) $t \sqsubseteq s$, b) $t' \sqsubseteq s$ and c) $\mathcal{P} \vdash_{\text{CRWL}} e \rightarrow s$ by induction on the depth l of a CRWL -proof for $e \rightarrow t$:

$\underline{l = 0}$ Three possible inference rules:

- **BT**. Then $t = \perp$ and defining $s = t'$ we have: a) $\perp \sqsubseteq s$, b) $t' \sqsubseteq s$ and c) $\mathcal{P} \vdash_{\text{CRWL}} e \rightarrow s$ (by $(R2)$).
- **RF**. Then the proof for $(R1)$ must be of the form $X \rightarrow X$, and hence $e = X$ and $t = X$. Then t' only can be X or $\perp$ (otherwise no CRWL inference could be applied and (R2) would not hold). We define s as X and then: a) $t \sqsubseteq X$ b) $t' \sqsubseteq X$ c) $\mathcal{P} \vdash_{\text{CRWL}} e \rightarrow s$ by $(R1)$.
- **DC**. Then $e = c$, $t = c$, with $c \in DC^0$. Then t' must be either c or $\perp$. In any case defining s as c the result holds.

$\underline{l > 0}$ There are three possible inference rules applied at the first step of the proof:

- **DC**. Then $e = c\ e_1\ \ldots\ e_m$, $\ t = c\ t_1\ \ldots\ t_m$ with $c \in DC^n \cup FS^{n+1}$, $\quad m \leq n$. Analogously $t' = c\ t_1\ \ldots\ t_m$ and the first inference rules of any proof for $(R1)$ y $(R2)$ must be of the form:

$$(R1) : \frac{e_1 \rightarrow t_1\ \ldots\ \ e_m \rightarrow t_m}{c\, e_1\ \ldots\ e_m \rightarrow c\, t_1\ \ldots\ t_m} \qquad (R2) : \frac{e_1 \rightarrow t'_1\ \ldots\ \ e_m \rightarrow t'_m}{c\, e_1\ \ldots\ e_m \rightarrow c\, t'_1\ \ldots\ t'_m}$$

  The proofs for $\mathcal{P} \vdash_{\text{CRWL}} e_i \rightarrow t_i$ and $\mathcal{P} \vdash_{\text{CRWL}} e_i \rightarrow t'_i$ have a maximum depth of $l-1$. Therefore by induction hypotheses exists $s_i \in Cterm_\perp$ satisfying $t_i, t'_i \sqsubseteq s_i$, and $\mathcal{P} \vdash_{\text{CRWL}} e_i \rightarrow s_i$ for all $1 \leq i \leq m$. Then defining $s = c\ s_1\ \ldots\ s_m$, $t \sqsubseteq s$, $t' \sqsubseteq s$ hold and $\mathcal{P} \vdash_{\text{CRWL}} e \rightarrow s$ with a proof starting with a DC inference.
- **JN**. Very similar to the previous case.
- **AF**. Then e is of the form $f\ \bar{e}_n$ with $e_i \in CTerm_\perp$ for $i = 1 \ldots n$. Moreover n is greater of equal to the program arity of f. Hence an AF inference must have been applied at the first step of any proof of (R2). In each case a suitable instance $(I1)$ y $(I2)$ must have been used. We call θ and θ' to the substitutions associated to the first and to the second instance respectively, $\theta, \theta' \in CSubst_\perp$.
  The first inference step of each proof will be of the following form:

$$(1) : \frac{e_1 \to t_1\theta, \ldots, e_k \to t_k\theta,\ C\theta,\ r\theta \to a,\ a\, e_{k+1} \ldots e_n \to t}{f\, e_1 \ldots e_k\, e_{k+1} \ldots e_n \to t}$$

$$(2) : \frac{e_1 \to t'_1\theta', \ldots, e_k \to t'_k\theta',\ C'\theta',\ r'\theta' \to a',\ a'\, e_{k+1} \ldots e_n \to t'}{f\, e_1 \ldots e_k\, e_{k+1} \ldots e_n \to t'}$$

with $(k > 0)$, t, $t' \neq \bot$ and the rule instances:

I_1: $(f\, t_1 \ldots t_k \to r \Leftarrow C)\theta \in [R]_\bot$ I_2: $(f\, t'_1 \ldots t'_k \to r' \Leftarrow C')\theta' \in [R]_\bot$

Now we consider separately two cases: a) I_1 e I_2 correspond to the same program rule, and b) each instance correspond to a different program rule.

a) Assume that I_1, I_2 Correspond to the same program rule, i.e. $t_i = t'_i$, $r = r'$, $C = C'$. We look for some $s \in CTerm_\bot$ such that $t \sqsubseteq s$, $t' \sqsubseteq s$, and also for some $\theta'' \in CSubst_\bot$ that allows us to build a CRWL proof for $\mathcal{P} \vdash_{\text{CRWL}} f\bar{e}_n \to s$ starting with a first inference rule application of the form

$$(3) : \frac{e_1 \to t_1\theta'', \ldots, e_k \to t_k\theta'',\ C\theta'',\ r\theta'' \to a'',\ a''\, e_{k+1} \ldots e_n \to s}{f\, e_1 \ldots e_k\, e_{k+1} \ldots e_n \to s}$$

We must check that it is possible to find CRWL -proofs for all the premises in (3). From the premises of (1), (2) we know that $\mathcal{P} \vdash_{\text{CRWL}} e_i \to t_i\theta$ and $\mathcal{P} \vdash_{\text{CRWL}} e_i \to t_i\theta'$ for each $i = 1 \ldots k$. Then by induction hypotheses, for every $i = 1 \ldots k$ exists a term s_i s.t. a) $t_i\theta \sqsubseteq s_i$, b) $t_i\theta' \sqsubseteq s_i$ and c) $\mathcal{P} \vdash_{\text{CRWL}} e_i \to s_i$. By a) and the lemma **??** exists also a substitution θ_i s.t. $t_i\theta_i = s_i$. Since $(t_1, \ldots, t_k)$ is linear, we can define θ'' as:

$$\theta''(X) = \begin{cases} \theta_i(X), & \text{if } X \in var(t_i) \text{ for some i} \\ \theta(X), & \text{otherwise} \end{cases}$$

By the definition $s_i = t_i\theta''$ and therefore the premises $e_i \to t_i\theta''$ of (3) admit a CRWL proof for all $i = 1 \ldots k$.

The condition C is of the form $d_1 == d_2$, with $d_1, d_2 \in Term$. We know by (1) that $C\theta$ has a CRWL proof, which means that each $d_1 == d_2$ has a CRWL -proof, which must be of the form:

$$\frac{d_1\theta \to u,\ d_2\theta \to u}{d_1\theta == d_2\theta} \quad u \in CTerm$$

Since $\theta \sqsubseteq \theta''$, we have by Lemma 5.3.ii) that $d_1\theta'' \to u$ and $d_2\theta'' \to u$ have CRWL proofs. Therefore the premise $C\theta''$ of (3) also has a CRWL -proof. Now we observe that $\theta(X) \sqsubseteq \theta''(X)$ and $\theta'(X) \sqsubseteq \theta''(X)$ for $X \in var(\bar{t}_n)$. From the premises of (1) and (2) we have: $\mathcal{P} \vdash_{\text{CRWL}} r\theta \to a$ and $\mathcal{P} \vdash_{\text{CRWL}} r\theta' \to a'$. Since in addition $var(r) \subseteq var(\bar{t}_n)$, by Lemma 5.3.ii) we can have that $\mathcal{P} \vdash_{\text{CRWL}} r\theta'' \to a$, $\mathcal{P} \vdash_{\text{CRWL}} r\theta'' \to a'$.

Then by induction hypotheses, there must exist some $a'' \in CTerm_\perp$ c s.t.: a) $a \sqsubseteq a''$, b) $a' \sqsubseteq a''$ and c) $\mathcal{P} \vdash_{\text{CRWL}} r\theta'' \to a''$, which crwl-proves the premise $r\theta'' \to a''$ in (3). Moreover since we have $\mathcal{P} \vdash_{\text{CRWL}} a\, e_{k+1} \ldots e_n \to t$, $\mathcal{P} \vdash_{\text{CRWL}} a'\, e_{k+1} \ldots e_n \to t'$ we have from Lemma 5.3, item iii): $\mathcal{P} \vdash_{\text{CRWL}} a''\, e_{k+1} \ldots e_n \to t$, $\mathcal{P} \vdash_{\text{CRWL}} a''\, e_{k+1} \ldots e_n \to t'$, and by induction hypotheses there exists some $s \in CTerm_\perp$ such that a) $t \sqsubseteq s$, b) $t' \sqsubseteq s$ c) $\mathcal{P} \vdash_{\text{CRWL}} a''\, e_{k+1} \ldots e_n \to s$, which proves the last premise of (3): $a''\, e_{k+1} \ldots e_n \to s$.

b) Now assume that I_1, I_2 are instances of two different program rules. In such case by the non-ambiguity criterion (remember that f is occurring in e and therefore by hypotheses is non-ambiguous) there exists γ=m.g.u. $(f\,\bar{t}_k, f\,\bar{t'}_k)$, i.e. $t_i\gamma = t'_i\gamma$ for $i = 1 \ldots k$ and $r\gamma = r'\gamma$. Calling u_i to $t_i\gamma = t'_i\gamma$, the rule instances can be seen as:

$$(f\, u_1 \ldots u_k \to r'' \Leftarrow C\gamma) \quad \text{and} \quad (f\, u_1 \ldots u_k \to r'' \Leftarrow C'\gamma).$$

Now we must look for some $s \in CTerm_\perp$ such that: a) $t \sqsubseteq s$, b) $t' \sqsubseteq s$ and c) $\mathcal{P} \vdash_{\text{CRWL}} f\bar{e}_n \to s$ for some substitution θ''. The proof of c) can be of one of these two forms

$$(4):\ \frac{e_1 \to u_1\theta'', \ldots, e_k \to u_k\theta'',\ C\gamma\theta'',\ r''\theta'' \to a'',\ a''\, e_{k+1} \ldots e_n \to s}{f\, e_1 \ldots e_k\, e_{k+1} \ldots e_n \to s}$$

$$(5):\ \frac{e_1 \to u_1\theta'', \ldots, e_k \to u_k\theta'',\ C'\gamma\theta'',\ r''\theta'' \to a'',\ a''\, e_{k+1} \ldots e_n \to s}{f\, e_1 \ldots e_k\, e_{k+1} \ldots e_n \to s}$$

We observe that γ unifies the heads and fusions the right-hand sides, but it doesn't relation C y C'. We consider the form (4) (the (5) is analogous). From the premises of (1) y (2) we know that $\mathcal{P} \vdash_{\text{CRWL}} e_i \to t_i\theta$ and $\mathcal{P} \vdash_{\text{CRWL}} e_i \to t'_i\theta'$ for $i = 1 \ldots k$. By induction hypotheses exists $s_i \in CTerm_\perp$ s.t.: a)$t_i\theta \sqsubseteq s_i$, b) $t'_i\theta' \sqsubseteq s_i$, and c) $\mathcal{P} \vdash_{\text{CRWL}} e_i \to s_i$. Since t_i, t'_i are unified by γ, we can apply the Lemma 5.1. Then there exist substitutions θ_i which we can restrict to the variables in u_i s.t. $u_i\theta_i = s_i$. $(u_1, \ldots, u_k)$ is a linear tuple because $(t_1, \ldots, t_k)$ and $(t'_1, \ldots, t'_k)$ are both linear. Then we can define a substitution θ'' as:

$$\theta''(X) = \begin{cases} \theta_i(X) & \text{if } X \in var(t_i, t'_i) \text{ for some } i,\ 1 \le i \le k \\ \theta(X) & \text{otherwise} \end{cases}$$

ensuring that there exist CRWL -proofs of $e_i \to u_i\theta''$ for all $i = \{1, \ldots, k\}$ in (4) (this is because $u_i\varphi = u_i\theta''$).

Checking that rest of the premises of (4) also have CRWL -proof requires similar arguments to those employed in the case a) and we omit the details.

□