Compiling a Functional Logic Language:
The Fair Scheme[⋆]

Sergio Antoy and Andy Jost

Computer Science Dept., Portland State University, Oregon, U.S.A.

antoy@cs.pdx.edu
andrew.jost@synopsys.com

Abstract. We present a compilation scheme for a functional logic programming language. The input program to our compiler is a constructor-based graph rewriting system in a non-confluent, but well-behaved class. This input is an intermediate representation of a functional logic program in a language such as Curry or $\mathcal{TOY}$. The output program from our compiler consists of three procedures that make recursive calls and execute both rewrite and pull-tab steps. This output is an intermediate representation that is easy to encode in any number of programming languages. Our design evolves the *Basic Scheme* of Antoy and Peters by removing the "left bias" that prevents obtaining results of some computations—a behavior related to the order of evaluation, which is counter to declarative programming. The benefits of this evolution are not only the *strong completeness* of computations, but also the *provability* of non-trivial properties of these computations. We rigorously describe the compiler design and prove some of its properties. To state and prove these properties, we introduce novel definitions of "need" and "failure." For non-confluent constructor-based rewriting systems these concepts are more appropriate than the classic definition of need of Huet and Levy.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features — Control structures; D.3.4 [*Programming Languages*]: Processors — Compilers; F.4.2 [*Mathematical Logic and Formal Languages*]: Grammars and Other Rewriting Systems — ; G.2.2 [*Discrete Mathematics*]: Graph Theory — Graph algorithms; F.1.2 [*Computation by Abstract Devices*]: Modes of Computation — Alternation and nondeterminism;

General Terms Languages, Non-Determinism, Graph, Rewriting, Compilation.

Keywords Functional Logic Programming Languages, Non-Determinism, Graph Rewriting Systems, Compiler Construction.

1 Introduction

Recent years have seen a renewed interest in the implementation of functional logic languages [16, 19, 24]. The causes of this trend, we conjecture, include the maturity of the paradigm [1, 5, 26], its growing acceptance from the programming languages community [6, 13, 29], and the discovery of and experimentation with new techniques [7, 9, 20] for handling the most appealing and most problematic feature of this paradigm—non-determinism.

[⋆] This material is based in part upon work supported by the National Science Foundation under Grant No. 1317249.

Non-determinism can simplify encoding difficult problems into programs [6, 10], but it comes at a price. The compiler is potentially more complicated and the execution is potentially less efficient than in deterministic languages and programs. The first issue is the focus of our work, whereas the second one is addressed indirectly. In particular, we present an easy to implement, *deterministic* strategy for *non-deterministic* computations. Our strategy is the only one to date in this class with a *proof* of its correctness and optimality.

Section 2 defines the *source* programs taken by our compiler as a certain class of non-confluent constructor-based graph rewriting systems. It also introduces a novel concept of "need" appropriate for these programs. Section 3 formally defines and informally describes the design of our compiler by means of three abstract *target* procedures that can be easily implemented in any number of programming languages. Section 4 relates to each other *source* and *target* computations and states some properties of this relation. In particular, it proves that every step executed by the *target* program on an expression is needed to compute a value of that expression in the *source* program. Section 5 formalizes and informally proves the strong completeness of our scheme: any value of an expression computed by the *source* program is computed by *target* program as well. Section 6 briefly outlines an on-going implementation in C++ of our scheme. Sections 7 and 8 summarize related work and offer our conclusion.

2 Background

We assume familiarity with constructor-based, many-sorted graph rewriting systems [23, 47]. It would be impossible to adequately summarize this notion within the boundaries of this paper, thus we only highlight some key points relevant to our discussion. An *expression* is a finite, acyclic, single-root graph defined in the customary way [23, Def. 2]. As usual, nodes are decorated by labels and successors. The signature contains a distinguished symbol called *choice*, defined later. A node labeled by this symbol has an additional decoration, a *choice identifier* [7, Def. 1]. Later, we will show how this identifier, e.g., an arbitrary integer, tracks a *choice* in an expression being evaluated. A graph *homomorphism* [23, Def. 10] is a mapping from nodes to nodes that preserves sorts, roots and, except for nodes labeled by variables, labels and successors. A *rewrite rule* is a pair of expressions in which the left-hand side is a pattern and every variable in the right-hand side occurs in the left-hand side as well. *Variables* occur only in rewrite rules. Therefore, we exclude narrowing computations from our discussion. We will readdress this apparent limitation later. A *rewriting computation* (also called *derivation*) is a finite or infinite sequence $e_0 \to e_1 \to \ldots$ in which $e_i \to e_{i+1}$ is a *rewrite step* [23, Def. 23]. Only rewrite steps are allowed in this definition. The *target* procedures output by our compiler, described in Sect. 3, execute a second kind of step called pull-tab. Later, we will relate to each other computations with and without steps of this second kind. At times, we will consider the reflexive closure of the one-step relation and will call "*null*" the step that does not perform any replacement. Every expression in a computation of e is called a *state of the computation* of e.

The class of rewrite systems that we compile is crucial for the relative simplicity, efficiency and provability of our design. Below we both describe and motivate this

class. Functional logic programming languages, such as Curry [28, 31] and $\mathcal{TOY}$ [22, 43], offer to a programmer a variety of high-level features including expressive constructs (e.g., list comprehension), checkable redundancy (e.g., declaration of types and free variables), visibility policies (e.g., modules and nested functions), and syntactic sugaring (e.g., infix operators, anonymous functions).

A compiler typically transforms a program with these high-level features into a program that is semantically equivalent, i.e., it has the same I/O behavior, but is in a form that is easier to compile and/or execute. This transformed program, which is the input of our compilation scheme, is a graph rewriting system in a class that we call *LOIS* (limited overlapping inductively sequential). The concept of definitional tree [2, 5], recalled below, characterizes this class.

A definitional tree is a hierarchical organization of the rewrite rules defining certain operations of a program. We use standard notations, in particular, if t and u are expressions and p is a node of t, then $t|_p$ is the *subexpression* of t rooted at p [23, Def. 5] and $t[p \leftarrow u]$ is the *replacement* by u of the subexpression of t rooted by p [23, Def. 9].

Definition 1. *$\mathcal{T}$ is a* partial definitional tree*, or* pdt*, if and only if one of the following cases holds:*

- $\mathcal{T} = branch(\pi, o, \bar{\mathcal{T}})$, *where π is a pattern, o is a node, called* inductive*, labeled by a variable of π, the sort of $\pi|_o$ has constructors $c_1, \ldots, c_k$ in some arbitrary, but fixed, ordering, $\bar{\mathcal{T}}$ is a sequence $\mathcal{T}_1, \ldots, \mathcal{T}_k$ of* pdt*s such that for all i in $1, \ldots, k$ the pattern in the root of $\mathcal{T}_i$ is $\pi[o \leftarrow c_i(x_1, \ldots, x_n)]$, where n is the arity of c_i and $x_1, \ldots, x_n$ are fresh variables.*
- $\mathcal{T} = rule(\pi, l \rightarrow r)$, *where π is a pattern and $l \rightarrow r$ is a rewrite rule such that $l = \pi$ modulo a renaming of variables and nodes.*
- $\mathcal{T} = exempt(\pi)$, *where π is a pattern.*

Definition 2. *$\mathcal{T}$ is a* definitional tree *of an operation f if and only if $\mathcal{T}$ is a* pdt *with $f(x_1, \ldots, x_n)$ as the pattern argument, where n is the arity of f and $x_1, \ldots, x_n$ are fresh variables.*

Definition 3. *An operation f of a rewrite system $\mathcal{R}$ is* inductively sequential *if and only if there exists a definitional tree $\mathcal{T}$ of f such that the rules contained in $\mathcal{T}$ are all and only the rules defining f in $\mathcal{R}$.*

Exempt nodes occur in trees of incompletely defined operations only. E.g., the definitional tree of the operation, **`head`**, that computes the first element of a list has an exempt node with pattern **`head []`**. Patterns do not need explicit representation in a definitional tree, but often their presence simplifies the discussion.

Definitional trees characterize the programs accepted by our compiler—every operation, except one, of these programs is inductively sequential. Inductive sequentiality is a syntactic property of a rewrite system. A simple algorithm for constructing, when it exists, a definitional tree from the rules defining an operation, and thus proving its inductive sequentiality, is in [5]. A feature-rich functional logic program is eventually stripped of its features and transformed into a graph rewriting system in the class defined in Def. 4. The details of this transformation are quite complex and include lambda

lifting [37], elimination of partial applications and high-order function [48], elimination of conditions [4], replacement of non-inductively sequential functions with inductively sequential ones [4] and replacement of logic (free) variables with generator functions [11, 40].

Definition 4 (LOIS). *A* LOIS *system is a constructor-based graph rewriting system $\mathcal{R}$ in which every operation of the signature of $\mathcal{R}$ either is the binary* choice *operation denoted by the infix symbol "*`?`*" and defined by the rules:*

```
x ? _ = x
_ ? y = y
```
(1)

or is inductively sequential. A LOIS *system will also be called a* source *program.*

All the non-determinism of a *LOIS* system is confined to the *choice* operation, which is also the only non-inductively sequential operation. While its rules can be used in a rewriting computation, the code generated by our compiler will not (explicitly) apply these rules. The reason is that the application of a rule of (1) makes an irrevocable decision in a computation. In this event, the completeness of computations can be ensured by techniques such as backtracking or copying which have undesirable aspects [7]. By avoiding the application of the *choice* rules, pull-tabbing (also bubbling [8, 9]) makes no irrevocable decisions.

LOIS systems have been widely investigated. Below we recall some key results that justify and support our choice of *LOIS* systems as the *source* programs of our compiler.

1. Any *LOIS* system admits a complete, sound and optimal evaluation strategy [3].
2. Any constructor-based conditional rewrite system is semantically equivalent to a *LOIS* system [4].
3. Any *narrowing* computation in a *LOIS* system is semantically equivalent to a *rewriting* computation in another similar *LOIS* system [11].
4. In a *LOIS* system, the order of execution of disjoint steps of an expression does not affect the value(s) of the expression [3, 12].

LOIS systems are an ideal core language for functional logic programs because they are general enough to perform any functional logic computation [4] and powerful enough to compute by simple rewriting [11, 40], without wasting steps [3] and without concerns about the order of evaluation [3, Lemma 20].

In particular, our decision of banning free (unbound) variables from our model is justified by [11, 40]. We will discuss in Sect. 7 a crucial difference between the strategy of [3] and the strategy implicitly defined by the *target* procedures.

The seminal concept of *needed redex* introduced in [34] for orthogonal term rewriting systems is inapplicable to and inappropriate for *LOIS* systems. *LOIS* systems are not orthogonal because the rules of *choice* overlap—an essential condition to provide the expressive power sought in modern functional logic languages through non-determinism. *LOIS* systems are constructor-based—reducing an expression to a *normal form* (an expression that has no steps) is interesting only when this normal form is a *value* (an expression in which every node of e is labeled by a constructor symbol). Values are normal forms, but there are normal forms that are not values, e.g., `head []`.

Such expressions are regarded as failing computations. A more general and precise definition of this concept will be provided shortly.

Below, we propose a novel definition of *need* which is better suited for our class of systems. This notion enables us to address the theoretical efficiency of a computation in *LOIS* systems much in the same way as the classic notion does in orthogonal systems.

Definition 5 (Needed). *Let S be a* source *program, e an expression of S whose root node we denote by p, and n a node of e. Node n is* needed for e*, and similarly* needed for p*, iff in any derivation of e to a constructor-rooted form the subexpression of e at n is derived to a constructor-rooted form. A node n (and the redex rooted by n, if any) of a state e of a computation in S is* needed *iff it is needed for some maximal operation-rooted subexpression of e. A computation $A : e_0 \to e_1 \to \cdots$ of some expression e_0 in S is* needed *iff it reduces only needed redexes.*

Our notion of need is a relation between two nodes (we also consider the subexpressions rooted by these nodes since they are in a bijection with the nodes). Our relation is interesting only when both nodes are labeled by operation symbols. If e is an expression whose root node p is labeled by an operation symbol, then p is trivially needed for p. This holds whether or not e is a redex and even when e is *already* a normal form, e.g., `head []`. In particular, *any* expression that is not a value has pairs of nodes in the needed relation. Finally, our definition is concerned with reaching a *constructor-rooted* form, not a *normal* form. Situations where a node n, root of an *irreducible* expression, is needed for an expression e enable aborting a possibly non-terminating computation of e which cannot produce a value. The next definition formalizes this point. An example will follow.

Definition 6 (Failure). *Let S be a* source *program and e an operation-rooted expression of S. Expression e is a* failure *iff there exists no derivation of e to a constructor-rooted form. When e is a failure, we may denote it with the symbol "$\bot$" instead of e if the nodes, labels, and other components of e are of no interest.*

In general, telling whether an expression e is a failure is undecidable, since it entails knowing whether some computation of e terminates. However, detecting failures in programming is common place. Indeed, in many programming languages a failure goes by the name of *exception*, a name that also denotes the mechanism for recovering from computations failing to produce a value. In functional logic programming, because of non-determinism, there are useful programming techniques based on failing computations [10] and failures are simply and silently ignored. Our notion of need makes detecting some failures easy, even in the presence of non-terminating computations. For example, consider the expression $e =$ `loop+(1/0)`, where `loop` is defined below and the other symbols have their usual meaning:

```
loop = loop                                                    (2)
```

It is immediate to see that the only redex of e is `loop` and consequently the computation of e does not terminate. Relying on the intuitive meaning of the symbols, since we have not defined them by rewrite rules, `1/0` is a failure and its root is needed for e. Hence,

e itself is a failure. Thus, the computation of e can be terminated (in a failure) even though e is reducible and `loop` is a needed redex, in the classic sense [34], of e.

The definition of the compiler in Fig. 1 rewrites failures to the distinguished symbol "$\bot$". These rewrites are only a notational convenience to keep the presentation compact. An implementation needs not rewrite failures to the "$\bot$" symbol. Instead, the internal representation of a node may be tagged to say whether that node is the root of a failure. We will show that a failure propagates from a node n to a node p, when n is needed for p. Rewriting an expression e to $\bot$ can be interpreted as recording (without performing a step) that e cannot be derived to a value.

We now explore some properties of our newly introduced notion of "need". An interesting aspect is its transitivity, which will become useful to prove some facts about computations.

Lemma 1 (Transitivity). *Let S be a* source *program, e an expression of S, e_1, e_2 and e_3 subexpressions of e such that n_i is the root of e_i and the label of n_i is an operation, for $i = 1, 2, 3$. If n_3 is needed for n_2 and n_2 is needed for n_1, then n_3 is needed for n_1.*

Proof. By hypothesis, if e_3 is not derived to a constructor-rooted form, e_2 cannot be derived to a constructor-rooted form, and if e_2 is not derived to a constructor-rooted form, e_1 cannot be derived to a constructor-rooted form. Thus, if e_3 is not derived to a constructor-rooted form, e_1 cannot be derived to a constructor-rooted form. □

Our notion of need generalizes the classic notion [34] with the difference that in orthogonal systems a redex has only one replacement, whereas in our programs a needed node may or may not root a redex. When it roots a redex, it may have more than one replacement and some replacement may or may not contribute to the computation of a value.

Lemma 2 (Extension). *Let S be a* source *program and e an expression of S derivable to a value. Assume that the* choice *symbol occurs neither in e nor in the right-hand side of any rule of S. If e' is an outermost operation-rooted subexpression of e, and n is both a node needed for e' and the root of a redex r, then r is a needed redex of e in the sense of* [34].

Proof. First we show that it is meaningful to consider the classic notion of need in the hypothesis of the claim. Since the *choice* symbol is banned from both the program and the top-level expression, it can be eliminated from S without changing any computation of e. S without "`?`" is inductively sequential and consequently orthogonal, actually strongly sequential [32]. Since e' is an outermost operation-rooted subexpression of e, the path from the root of e to the root of e' excluded consists of nodes labeled by constructor symbols. Hence, e can be derived to a value only if e' is derived to a value and e' can be derived to a value only if e' is derived to a constructor rooted form. By assumption, in any derivation of e' to a constructor rooted form r is derived to a constructor rooted form, hence it is reduced. Thus, r is a needed redex of e according to [34]. □

We close this section with some auxiliary results that shed some light on the use of definitional trees and are instrumental in proofs of later claims.

Lemma 3 (Rule selection). *Let S be a* source *program, e an expression of S rooted by a node n labeled by some operation f and $\mathcal{T}$ a definitional tree of f. If $\mathcal{T}_1$ is a node of $\mathcal{T}$ with pattern π, $h(\pi) = e$ for some match h, and $l \to r$ is a rule that reduces a state of a computation of e at n, then $l \to r$ is in a leaf of $\mathcal{T}_1$, including $\mathcal{T}_1$ itself if $\mathcal{T}_1$ is a leaf.*

Proof. Rule $l \to r$ is in a leaf of $\mathcal{T}$, since these are all and only the rules defining f. We prove that if $l \to r$ is not in a leaf of $\mathcal{T}_1$, then it cannot reduce e at n. Since n is the root of e, there exists at most one reduction at n in any computation of e. As in any proof comparing graphs, equality is intended modulo a renaming of nodes [23, Def. 15]. Let $\mathcal{T}_2$ be a node of $\mathcal{T}$ disjoint from $\mathcal{T}_1$ and $\mathcal{T}_0$ the closest (deepest in $\mathcal{T}$) common ancestor of $\mathcal{T}_1$ and $\mathcal{T}_2$. Let o_0 be the inductive node $\mathcal{T}_0$, and $h(o) = p$ for some node p of e. By Def. 1 $\mathcal{T}_1 = \mathcal{T}_0[o_0 \leftarrow c_1(\ldots)]$, where c_1 is a constructor symbol labeling some node o_1 and the arguments of c_1 do not matter. Likewise, $\mathcal{T}_2 = \mathcal{T}_0[o_0 \leftarrow c_2(\ldots)]$, where c_2 is a constructor different from c_1 labeling some node o_2. Since π matches e, the label of p is c_1. In e, every node in a path from n (excluded) to p is labeled by a constructor. Hence, the same nodes with the same labels persist in every state of the computation of e that does not replace n. Let π' be a pattern of a rule in a leaf of $\mathcal{T}_2$. Pattern π' can never match a state of the computation of e, say e', in which n was not replaced because any homomorphism of such a match would have to map o_2, which is labeled by c_2, to p, which is labeled by c_1, and by construction $c_1 \neq c_2$. □

Lemma 4 (Needed). *Let S be a* source *program, e an expression of S rooted by a node n labeled by some operation f and $\mathcal{T}$ a definitional tree of f. If $\mathcal{T}_1$ is a* branch *node of $\mathcal{T}$ with pattern π and inductive node o, $h(\pi) = e$ for some match h, and $h(o) = p$, for some node p of e labeled by an operation symbol, then p is needed for n.*

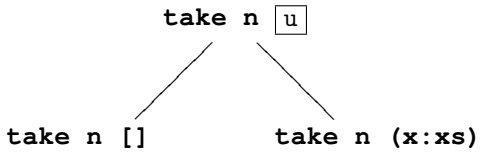
Proof. By Lemma 3 any rule reducing any state of a computation of e at the root is in a leaf of $\mathcal{T}_1$. Let $l \to r$ be a rule in a leaf of $\mathcal{T}_1$. By Def. 1, l is an instance of π, i.e., $l = \sigma(\pi)$, for some homomorphism σ. Since o is the inductive position of π in $\mathcal{T}_1$, every child of $\mathcal{T}_1$ has a pattern of the form $\pi[o \leftarrow c(x_1, \ldots, x_n)]$, where c is a constructor. Thus, in l, $\sigma(o)$ is a node labeled by a constructor. Every node of e in a path from the root n to p, end nodes excluded, is labeled by a constructor. This condition persists in any state of a computation of e that does not reduce e at n. Unless $e|_p$ is reduced to a constructor rooted expression, l cannot match any state of a computation of e and hence e cannot be reduced at the root. Thus, by Def. 5, p is needed for n. □

We present a small example to see the above results in action. Two functions play a major role in the example: **++**, a function that concatenates two lists, and **take**, a function that takes a prefix of a given length of a list:

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs++ys)

take _ [] = []
take n (x:xs) =
     if n==0 then []
     else x : take (n-1) xs
```
(3)

The definitional tree of **take** is shown below. To ease readability, we show only the patterns of *rule* nodes, but not the rules themselves, and we box inductive variables:



The evaluation of $e =$ **take 2 ([1]++[2,3])** goes as follows. Expression e is matched by the pattern in the root node of the definitional tree of **take** which is a *branch*. The expression $t =$ **[1]++[2,3]** is matched by variable u, which is inductive. Hence, by Lemma 4, t is needed for e. The evaluation of t to a constructor rooted expression produces **1:([]++[2,3])**. The resulting state of the computation of e is $e' =$ **take 2 (1:([]++[2,3]))**. Expression e' is matched the right leaf of the definitional tree of **take**. Since this is a *rule* node, e' is a redex. Lemma 3 confirms that the second rule of **take** is the only one that can reduce e'. Finally, Lemma 2 confirms that t is a needed redex in the classic sense.

3 Compilation

We describe the compilation of functional logic programs abstractly. The input of the compilation is a *LOIS* system S called the *source* program. We construct the definitional tree of every operation of S's signature except the *choice* operation. We compile both the signature of S and the set of definitional trees into three *target* procedures denoted **D** (Dispatch), **N** (Normalize) and **S** (Step). These procedures both make recursive calls, and execute *rewrite* [23, Def. 23] and *pull-tab* [7, Def. 2] steps. A concrete compiler only has to represent graphs as objects of some language L and map the *target* procedures into procedures (functions, methods, subroutines, etc.) of L that execute both the recursive calls and the replacements originating from the steps.

This style of compilation for functional logic languages was pioneered in [16], where also three procedures were defined for the same purpose. We will compare these two approaches in Section 7, but in short, our strategy handles failures, avoids "don't know" non-determinism, and ensures the (strong) completeness of computations. None of these properties holds for the scheme of [16].

A *pull-tab* step is a binary relation over the expressions of a *source* program similar to a rewrite step—in a graph a (sub)graph is replaced. The difference with respect to a rewrite step is that the replacement is not an instance of the right-hand side of a rewrite rule, but is obtained according to the following definition. Very informally, if e is an expression of the form $s(\ldots, x\texttt{?}y, \ldots)$, where s is not the *choice* symbol, then a pull-tab step of e produces $s(\ldots, x, \ldots)$ **?** $s(\ldots, y, \ldots)$. It seems very natural for pull-tab steps, as well, to call the (sub)graph being replaced the *redex*.

Definition 7 (Pull-tab). *Let e be an expression, n a node of e, referred to as the* target*, not labeled by the* choice *symbol and $s_1 \ldots s_k$ the successors of n in e. Let i be an index in $\{1, \ldots k\}$ such that s_i, referred to as the* source*, is labeled by the* choice *symbol and*

$\mathbf{D}(g;\bar{G}) =$
  case g **of**
    when x **?** y**:** $\mathbf{D}(\bar{G};x;y)$; **D**.1
    when $\bot$**:** $\mathbf{D}(\bar{G})$; **D**.2
    when g is a value**:** $\mathbf{D}(\bar{G})$; -- *yield* g **D**.3
    default: $\mathbf{N}(g)$**;**
      if $\mathbf{vn}(g)$ **then** $\mathbf{D}(\bar{G})$; **else** $\mathbf{D}(\bar{G};g)$; **D**.4
$\mathbf{D}(null) = null$; -- *program ends* **D**.5

$\mathbf{N}(c(\ldots,\bot,\ldots)) = null;\{$**return** *true*$\}$ **N**.1
$\mathbf{N}(c(\ldots,p\text{:}\,$**?**$(\_,\_),\ldots)) = \text{PULL}(p);\{$**return** *false*$\}$ **N**.2
$\mathbf{N}(c(x_1,\ldots,x_k)) = \mathbf{N}(x_1);\ldots;\mathbf{N}(x_k);$
$\{$**return** $\mathbf{vn}(x_1) \vee \ldots \vee \mathbf{vn}(x_k)\}$ **N**.3
$\mathbf{N}(n) = \mathbf{S}(n);\{$**return** *false*$\}$ **N**.4

---

compile $\mathcal{T}$
  case $\mathcal{T}$ **of**
    when $rule(\pi, l \to r)$**:**
      output $\mathbf{S}(l) = \text{REWR}(r)$; **S**.1
    when $exempt(\pi)$**:**
      output $\mathbf{S}(\pi) = \text{REWR}(\bot)$; **S**.2
    when $branch(\pi, o, \bar{\mathcal{T}})$**:**
      $\forall \mathcal{T}' \in \bar{\mathcal{T}}$ **compile** $\mathcal{T}'$
      output $\mathbf{S}(\pi[o \leftarrow \bot]) = \text{REWR}(\bot)$; **S**.3
      output $\mathbf{S}(\pi[o \leftarrow p\text{:}\,$**?**$(\_,\_)]) = \text{PULL}(p)$; **S**.4
      output $\mathbf{S}(\pi) = \mathbf{S}(\pi|_o)$; **S**.5
$\mathbf{S}(c(\ldots)) = null$ **S**.6

Fig. 1. Compilation of a *source* program with signature Σ into a *target* program consisting of three procedures: **D**, **N**, and **S**. The rules of **D** and **N** depend only on Σ. The rules of **S** are obtained from the definitional trees of the operations of Σ with the help of the procedure **compile**. The structure of the rules and the meaning of symbols and notation are presented in Def. 9. The notation $\mathbf{vn}(x)$ stands for the value returned by $\mathbf{N}(x)$. The symbol c stands for a generic constructor of the *source* program, $\bot$ is the fail symbol, and *choice* identifiers are used only within pull-tab steps hence they are not shown. A symbol of arity k is always applied to k arguments. Line comments, introduced by "--", indicate when a value should be yielded, such as to the read-eval-print loop of an interactive session, and where the computation end. The call to a *target* procedure with some argument g consistently and systematically operates on the *trace* of g. Hence, tracing is not explicitly denoted.

*let t_1 and t_2 be the successors of s_i in e. Let e_j, for $j = 1, 2$, be the graph whose root is a fresh node n_j with the same label as n and successors $s_1 \ldots s_{i-1} t_j s_{i+1} \ldots s_k$. Let $e' = e_1$ **?** e_2. The* pull-tab *of e with source s_i and target n is $e[n \leftarrow e']$ and we write $e \rightleftharpoons e[n \leftarrow e']$.*

A pull-tabbing computation of an expression e_0, denoted $e_0 \rightleftharpoons e_1 \rightleftharpoons \ldots$ generalizes a rewrite computation of e_0 by allowing any combination of both rewrites and pull-tabs.

Without some caution, this computation is unsound with respect to rewriting. Unsoundness may occur when some *choice* has two predecessors. For example, consider [18]:

```
xor x x where x = False ? True
```
(4)

A pictorial representation of this expression is in the left-hand side of Fig. 2. The *choice* of this expression is pulled up along two paths creating *two pairs* of strands, one for each path, which eventually must be pair-wise combined together. Some combinations will contain mutually exclusive alternatives, i.e., subexpressions that cannot be obtained by rewriting because they combine both the left and right alternatives of the same *choice*. Fig. 2 presents an example of this situation.

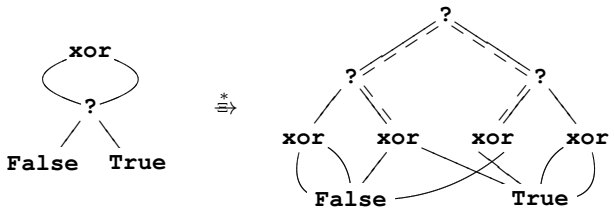


Fig. 2. *Pictorial representation of two states of the computation of* (4)*: the initial state to the left, and the state after three pull-tab steps to the right. Every* choice *in every state has the same identifier which is then omitted from the representation. The dashed paths are inconsistent, since they combine the left and right alternatives of the same* choice*, and therefore should be discarded.*

The soundness of pull-tabbing computations is preserved so long as the alternatives of a *choice* are never combined in the same expression [7]. To this aim, a node n labeled by the *choice* symbol is decorated with a *choice identifier* [7, Def. 1], such as an arbitrary, unique integer created when n is "placed in service" [7, Princ. 1]. When a *choice* is pulled up, this identifier is preserved. Should a *choice* be reduced to either of its alternatives, every other *choice* with the same identifier must be reduced to the same alternative. A very similar idea in a rather different setting is proposed in [18, 20]. A rewriting computation that for any *choice* identifier i consistently takes either the left or the right alternative of i is called a *consistent* computation. Furthermore, [7, Th. 1] shows that consistent computations with pull-tab steps are correct (i.e., sound and complete) with respect to rewriting computations.

The notion of *trace* [16], recalled below, allows us to keep track of a subgraph in a graph after the graph undergoes a sequence of replacements. The definition is non-trivial, but its application in an implementation is straightforward. We will discuss this point after defining the *target* procedures.

Definition 8 (Trace). *Let $g_0, g_1, \ldots$ be a sequence of expressions such that, for all $i > 0$, g_i is obtained from g_{i-1} by a replacement, i.e., there exist an expression r_{i-1} compatible* [23, Def. 6] *with g_{i-1} and a node p_{i-1} such that $g_i = g_{i-1}[p_{i-1} \leftarrow r_{i-1}]$. A node m of g_i is called a* trace *of a node n of g_j, for $j \leqslant i$, according to the following definition by induction on $i \geqslant 0$. Base case, $i = 0$: m is a trace of n iff $n = m$. Ind. case, $i > 0$: by assumption $g_i = g_{i-1}[p_{i-1} \leftarrow r_{i-1}]$ and by the induction hypothesis it is*

defined whether a node q of g_{i-1} is a trace of n. A node m of g_i is a trace of a node n of g_j iff there exists a trace q of n in g_{i-1} such that $m = q$ or m is the root of r_{i-1} and $q = p_{i-1}$.

Definition 9 (*Target* procedures). *Each* procedure *of the* target *system takes a graph, or sequence of graphs in the case of* **D***, as argument. Each procedure is defined by cases on its argument. Each case, called a* rule*, is selected by a liberal form of pattern matching and is defined by a possibly empty sequence of semicolon-terminated actions, where an* action *is either a recursive call to a* target *procedure, or a graph replacement* [23, Def. 9] *resulting from either a rewrite* [23, Def. 23] *or a pull-tab step* [7, Def. 2]*. In addition, procedure* **N** *returns a Boolean shown between curly braces in the pseudo-code. The rules are presented in Fig. 1. The rules have a priority as in common functional languages. Rules with higher priority come first in textual order. The application of a rule is allowed only if no rule of higher priority is applicable. Any reference to a node in the actions of any rule is the* trace [16] *of the node being referenced, i.e., tracing is consistently and systematically used by every rule without explicit notation. The notation* null *is a visible representation of an empty sequence of expressions, actions, steps, etc. depending on the context. The notations* $\text{REWR}(p)$ *and* $\text{PULL}(p)$ *are a rewrite and pull-tab steps, respectively, where p is the root of the replacement and the redex is the root of the argument of the rule where the notations occur. Graphs are written in* linear notation [23, Def. 4]*, e.g., in p:e, p is the root node of the pattern expression e, with the convention that nodes are explicitly written only when they need to be referenced.*

The *trace* [16] of t captures the changes that t undergoes as it passes through *target* procedures. An implementation in which the expression being evaluated is a global, persistent datum passed to the *target* procedures by reference provides very efficient tracing. Considering traces is essential for the correctness of our approach. In fact, some node in some graph may have two predecessors, hence the identity of this node must be preserved. For example, in rule **N**.3 of Fig. 1, the subgraphs x_1 and x_2 might be the same. Using traces preserves the identity of this node throughout a computation. Not only does this improve efficiency by avoiding repeated computations, it is essential to the soundness of computations. If the *same* non-deterministic expression is re-evaluated to a *different* value, the computation is unsound. Formalisms that might break this identity, e.g., because they look at expressions as trees instead of graphs, must introduce some device to preserve the identity. For example, CRWL [26] uses the call-time choice semantics [35] or let-constructs [42].

The *target* procedures are defined using pattern matching to select which rule of a procedure must be applied to the expression argument of the procedure. Since every pattern of every rule is linear—i.e., every variable occurs at most once in a pattern—no unification is necessary. A rule is selected by a simple chain of cases over the appropriate symbols of the argument. The notation $\bar{G}$, used in the definition of the rules of **D** in Fig. 1, stands for a sequence of zero or more objects. Whenever appropriate and understandable from the context, a single object may stand for a sequence containing only that object. Subsequences and/or individual objects in a sequence are separated by a semicolon. The empty sequence is denoted by "*null*". The *target* procedures execute only two particular kinds of replacement. The graph where the replacement occurs is

always the procedure argument and this argument is always the redex. Hence, we use the simpler notations introduced in Def. 9.

Procedure **D** manages a queue of expressions being evaluated. If the queue is not empty, it examines the expression, e, at the front of the queue. Depending on the form of e, e may be removed from queue or it may undergo some evaluation steps and be placed back at the end of the queue. Initially, the queue contains only the top-level expression. Pull-tabbing steps pull *choices* toward the root. If the front of the queue is a *choice*-rooted expression e, e is removed from the queue and its two alternatives are placed at the end of the queue (rule **D**.1). Their order does not matter because by Lemma 5 any call to **N** terminates. Therefore, any expression in the queue is a subexpression of a state of computation of the top-level expression. Since we use pull-tab steps, some of these expressions could be inconsistent. Thus, we will refine this rule, after introducing the notion of fingerprint, to discard inconsistent expressions. If the expression at the front of the queue is a failure, it is removed from the queue (rule **D**.2). If the expression at the front of the queue is a value, it is removed from the queue as well (rule **D**.3) after being yielded to a consumer, such as the read-eval-print loop of an interpreter. Finally, if no previous case applies, the expression e at the front of the queue is passed to procedure **N** that executes some steps of e (we will show a finite number) and returns whether the result should be either discarded or put back at the end of the queue (rule **D**.4). A result is discarded when it cannot be derived to a value. If the argument of **D** is the empty queue, the computation halts (rule **D**.5).

Procedure **N** either executes steps (of constructor-rooted expression), or invokes **S**. These steps do not depend on any specific operation of the *source* program. Like the other *target* procedures, the steps executed by **N** update the state of a computation. In addition to the other *target* procedures, **N** also returns a Boolean value. This Boolean value is true if and only if the expression argument of **N** cannot be derived to a value. This situation occurs when the argument e of an invocation of **N** is constructor-rooted, and an argument of the root is either a failure or (recursively) it cannot be reduced to a value (rule **N**.1). An example will follow shortly. If an argument of the root of e is a *choice*, then e undergoes a pull-tab step (rule **N**.2). The resulting reduct is a *choice* that procedure **D** will split it into two expressions. If e is constructor-rooted, and neither of the above conditions holds, then **N** is recursively invoked on each argument of the root (rule **N**.3). Finally, if the argument e of an invocation of **N** is operation-rooted, then procedure **S** is invoked on e (rule **N**.4) in hopes that e will be derived to a constructor-rooted expression and eventually one of the previous cases will be executed.

The following example shows why *target* procedures **N** cannot rewrite constructor-rooted expressions to $\bot$. In the following code fragment, **e** is a constructor-rooted expression that cannot be derived to a value, hence a failing computation, but not a failure in the sense of Def. 6. Let **snd** be the operation that returns the second component of a pair and consider:

```
t = e ? snd e where e = (⊥,0)
```
(5)

If **e** is rewritten to $\bot$, for some orders of evaluation **t** has no values, since **snd** $\bot$ is a failure. However, **0** is a value of **t**, since it is also a value of **snd (**$\bot$**,0)**.

Procedure **S** executes a step of an operation rooted expression. Each operation f of the *source* program contributes a handful of rules defining **S**. We call them $\mathbf{S}_f$*–rules*. The pattern (in the *target* program) of all these rules is rooted by f. Consequently, the order in which the operations of the *source* program produce **S**-rules is irrelevant. However, the order among the $\mathbf{S}_f$–rules is relevant. More specific rules are generated first and, as stipulated earlier, prevent the application of less specific rules. Let $\mathcal{T}$ be a definitional tree of f. At least one rule is generated for each node of $\mathcal{T}$. Procedure **`compile`**, which generates the $\mathbf{S}_f$–rules, visits the nodes of $\mathcal{T}$ in post-order. If π is the pattern of a node $\mathcal{N}$ of $\mathcal{T}$, the patterns in the children of $\mathcal{N}$ are instances of π. Hence, rules with more specific patterns textually occur before rules with less specific patterns. In the following account, let e be an f-rooted expression and the argument of an application of an $\mathbf{S}_f$-rule R and $\mathcal{N}$ the node of the definitional tree of f whose visit by **`compile`** produced R. If $\mathcal{N}$ is a *rule* node, then e is a redex and consequently reduced (rule **S**.1). If $\mathcal{N}$ is an *exempt* node, then e is a failure and it is reduced to $\perp$ (rule **S**.2). If $\mathcal{N}$ is a *branch* node, we have shown in Lemma 4 that unless p is reduced to a constructor-rooted expression, e cannot be reduced to a constructor-rooted expression. Thus, if p is a failure, e is a failure as well and consequently is reduced to $\perp$ (rule **S**.3). If p is a *choice*, e undergoes a pull-tab step (rule **S**.4). Finally, if p is operation-rooted, p becomes the argument of a recursive invocation of **S** (rule **S**.5). The last rule, labeled **S**.6, handles situations in which **S** is applied to an expression which is already constructor-rooted. This application occurs only to nodes that are reachable along multiple distinct paths, and originates only from rule **N**.3.

4 Properties

To reason about computations in the *target* program, we introduce some new concepts. A *call tree* is a possibly infinite, finitely branching tree in which a branch is a call to a *target* procedure whereas a leaf is a step in the *source* program. This concept offers a simple relation between computations in a *source* program and computations in the corresponding *target* program. If e is an expression of the *source* program, a left-to-right traversal of the *call tree* of $\mathbf{D}(e)$ visits the sequence of steps of a computation of e in the *source* program. In this computation, we allow pull-tab steps in addition to rewrite steps, but never apply a rule of *choice*.

Definition 10 (Call tree). *Let S be a* source *program and T the* target *program obtained from S according to the* Fair Scheme. *A* call tree *rooted by X, denoted $\Delta(X)$, is inductively defined as follows: if X is a null action or a rewrite or pull-tab step, we simply let $\Delta(X) = X$. If X is a call to a* target *procedure of T executing a rule with sequence of actions $X_1; \ldots X_n$, then $\Delta(X)$ is the tree rooted by X and whose children are $\Delta(X_1), \ldots \Delta(X_n)$. If e is an expression of S, then a left-to-right traversal of rewrite and pull-tab steps of $\mathbf{D}(e)$ is called the* simulated *computation of e and denoted $\omega(\mathbf{D}(e))$.*

The word "simulation" has been used in transformations of rewrite systems for compilation purposes [25, 38]. The name "simulated computation" stems from the property that, under the assumption of Def. 10, $\omega(\mathbf{D}(e))$ is indeed a pull-tabbing computation

of e in the *source* program. This will be proved in Cor. 2. We start with some preliminary results. *Choice* identifiers are ignored in the claims presented below. In other words, we disregard the fact that pull-tabbing creates inconsistent expressions. Inconsistent expressions should not be passed as arguments to procedures **S** and **N**. We will describe later how to ensure this condition, but for the time being we ignore whether an expression is consistent. An example of call tree is presented below.

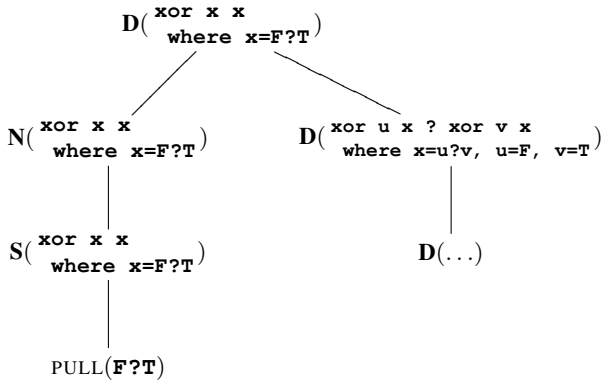


Fig. 3. *Topmost portion of the call tree of the expression defined in* (4)*. The syntax of expressions is Curry. The values* `False` *and* `True` *are abbreviated by* `F` *and* `T`*, respectively.*

Theorem 1 (Optimality). *Let S be a* source *program and* **S** *the step procedure of the corresponding* target *program. If e is an operation-rooted expression of S, then:*

1. $\mathbf{S}(e)$ *executes a replacement at some node n of e,*
2. *node n is needed for e,*
3. *if the step at n is the reduction to $\bot$, then e is a failure.*

Proof. We first describe the structure of the computation of $\mathbf{S}(e)$. By the rules of **S** in Fig. 1, in $\Delta(\mathbf{S}(e))$, $\mathbf{S}(e)$ has a single child that is either a step of S, when one of rules $\mathbf{S}.1$–$\mathbf{S}.4$ is applied to e, or a recursive invocation $\mathbf{S}(e|_o)$, for some node o of e different from the root of e, when rule $\mathbf{S}.5$ is applied to e. We will shortly prove that o is labeled by an operation. Thus, $\Delta(\mathbf{S}(e))$ is finite because at every recursive invocation its argument gets smaller and consists in a straight sequence of one or more invocations to **S** terminated by a step. Each claim of the theorem is proved by structural induction on $\Delta(\mathbf{S}(e))$. The base case is when there are no recursive invocations to **S**, i.e., the child of **S**(e) is a step of the *source* program. The inductive case is when there are recursive invocations to **S**.

1. Node n is witnessed by the root of the redex of e replaced by the step in the single leaf of $\Delta(\mathbf{S}(e))$.

2. If n is the root of e, the claim is trivial, since by assumption n is labeled by an operation. Otherwise, the child of $\mathbf{S}(e)$ in $\Delta(\mathbf{S}(e))$ is $\mathbf{S}(e|_o)$, where o is determined as follows. Expression e is rooted by some operation f. Let $\mathcal{T}$ be the definitional tree of f. There exists a *branch*, say $\mathcal{T}'$, of $\mathcal{T}$ with pattern π and inductive node o and a match h such that $h(\pi) = e$. We show that this condition implies that $e|_o$ is rooted by an operation symbol as well. By Definition 1, the root node of $\pi|_o$ is labeled by a variable. For each constructor c of the sort of this variable, there is a child of $\mathcal{T}'$ whose pattern is $\pi[o \leftarrow c(\ldots)]$, where the arguments of c are fresh variables. If the root node of $e|_o$ were labeled by a constructor, the pattern of some child of $\mathcal{T}'$ would match e, since the rules of $\mathbf{S}_f$ are generated by a post-order traversal of $\mathcal{T}$, hence the patterns of the children of $\mathcal{T}'$ are tried before the pattern in $\mathcal{T}'$. Thus, $e|_o$ is not rooted by a constructor symbol. By Lemma 4, node o is needed for e. By the induction hypothesis, there exists some node n such that $\mathbf{S}(e|_o)$ executes a step at n and n is needed for $e|_o$. By Lemma 1, the relation "*needed for*" is transitive. Thus, node n is needed for e as well.
3. Suppose that $\mathbf{S}(e)$ results in the step $e|_n \to \bot$. By the rules of $\mathbf{S}$ in Fig. 1, $e|_n$ is rooted by some operation f and is matched by the pattern of some *exempt* node of the definitional tree of f. By Lemma 3, there are no rules that can reduce any state of a computation of $e|_n$ at the root. Since $e|_n$ is operation rooted, it cannot be reduced to a constructor-rooted expression, hence it is a failure. By the previous point of this theorem, node n is needed for e, hence e is a failure. □

In passing, we observe that the acyclicity of expressions is instrumental to prove Theorem 1. For any expression e, $\Delta(\mathbf{S}(e))$ is finite because if $\mathbf{S}(e)$ makes a recursive invocation, the argument of this recursive invocation is a *proper* subexpression of e. If e had cycles, then the argument of the recursive invocation could be e itself. However, acyclicity is not necessary. In fact, [23, Def. 18] introduces the notion of *admissibility*. A term graph e is admissible if no node of a cycle of e is labeled by an operation. Admissibility is used to prove the confluence of a certain class of programs [23, Th. 1]. In our context, confluence is neither required nor desired. However, admissibility, which is weaker than acyclicity, is still sufficient to prove the termination of $\mathbf{S}$ and consequently all the claims of Theorem 1.

Theorem 1 is significant. The execution of $\mathbf{S}(e)$, for any operation-rooted expression e, terminates with a step. If the step is a rewrite to $\bot$, then e has no values. This knowledge is important to avoid wasting unproductive computational resources on e. If the step is a rewrite, then that rewrite is unavoidable. More precisely, if e has some value (a fact that generally *cannot* be known *before* obtaining a value), then we have to execute that rewrite to obtain a value of e. In this way, computational resources are conservatively used. If the step is a pull-tab, then reducing the *choice* source of the pull-tab is needed to reduce the redex target of the pull-tab to a constructor-rooted expression. Generally, we cannot know in advance which alternative of the *choice* might produce a value, hence both alternatives must be tried. This is exactly what pull-tabbing provides without committing to either alternative. In this case too, computational resources are not wasted.

Below we state some properties of the computation space of the *target* program that culminate in Corollary 2. The correctness of the *Fair Scheme* is a relatively straightforward consequence of this corollary.

Corollary 1 (N termination). *Let S be a* source *program and* **N** *the normalize procedure of the corresponding* target *program. For any expression e of S, the execution of* $\mathbf{N}(e)$ *terminates.*

Proof. We show that $\Delta(\mathbf{N}(e))$ is finite by structural induction on e. If e is matched by the pattern in rule **N**.1 or in rule **N**.2, $\mathbf{N}(e)$ has a single child which is a step. If e is matched by the pattern in rule **N**.3, the claim is a direct consequence of the induction hypothesis. The last rule, **N**.4, is intended only when e is operation-rooted. In this case, the claim is a consequence of Theorem 1. If e is not operation-rooted, then e is *choice*-root and $\mathbf{N}(e)$ has no execution. In passing, we note that, for any x, in $\Delta(\mathbf{D}(x)))$ **N** is never called with a *choice*-root argument. □

Lemma 5 (Space). *Let S be a* source *program,* **D** *the dispatch procedure of the corresponding* target *program, and e an expression of S. If $\Delta(\mathbf{D}(e))$ is infinite, then:*

1. *$\Delta(\mathbf{D}(e))$ has exactly one infinite path, say B;*
2. *B is rightmost in $\Delta(\mathbf{D}(e))$;*
3. *B contains all and only the applications of* **D** *in $\Delta(\mathbf{D}(e))$;*
4. *rule* **D**.4 *is applied an infinite number of times in B.*

Proof. If $b_0, b_1, \ldots$ is an infinite path in $\Delta(\mathbf{D}(e))$, then, for all i in $\mathbb{N}$, b_i is a call to **D**. In fact, if b_i were not a call to a *target* procedure, it would be a leaf in $\Delta(\mathbf{D}(e))$, and if b_i were a call to either **S** and **N**, then the call tree of b_i would be finite by Th. 1 and Cor. 1, respectively. For all $i > 0$, b_i is a recursive invocation of **D** resulting from the application of a rule of **D** since these are the only rules that invoke **D**. In all these rules, there is a single recursive invocation of **D**. This invocation is the last action in the right-hand side of the rule that produces the recursive invocation. By Def. 10, the recursive invocation is the rightmost child of b_{i-1}. Thus, the infinite path of $\Delta(\mathbf{D}(e))$ is the rightmost one, hence it is unique, and it consists exclusively of applications of **D**. In $\Delta(\mathbf{D}(e))$, there are no other applications of **D** outside this path. Now suppose that the rightmost path of $\Delta(\mathbf{D}(e))$ is infinite, but has a finite number of applications of rule **D**.4. There exists some k such that $b_k = \mathbf{D}(L_k)$ and, for $i > k$, $b_i = \mathbf{D}(L_i)$ is obtained from $\mathbf{D}(L_{i-1})$ with rule **D**.1, **D**.2, or **D**.3. The elements of L_i are the same as the elements of L_{i-1} except that one element l of L_{i-1} is either removed or replaced by two elements that both have fewer nodes than l itself. For all i, L_i is not the empty sequence, otherwise $\Delta(\mathbf{D}(e))$ would be finite. Therefore, for some $j > k$, the first element of L_j is an expression consisting of a single node, say p. Node p is labeled by an operation symbol, since constructors and failures would have been removed by rules **D**.2 and **D**.3. Hence, by the definition of **D** in Fig. 1, rule **D**.4 must be applied to $\mathbf{D}(L_j)$, contrary to the assumption. □

Lemma 6 (State subexpressions). *Let S be a* source *program,* **D** *the dispatch procedure of the corresponding* target *program, and e an expression of S. If $\mathbf{D}(L_0), \mathbf{D}(L_1), \ldots$ is the (finite or infinite) rightmost path of $\Delta(\mathbf{D}(e))$, then for every L_i in the path, the elements of L_i are subexpressions of a state of the computation of e.*

Proof. The proof is by natural induction on i. Base case: $i = 0$. By the Def. 10, L_0 is a queue containing only e which trivially is a state of the computation of e. Ind. case: $i > 0$. If L_i is obtained by applying a rule of **D** to $\mathbf{D}(L_{i-1})$, except rule $\mathbf{D}.4$, the claim is immediate from the definition of **D** in Fig. 1, under the induction hypothesis assumption that the claim holds for L_{i-1}. If L_i is obtained by applying rule $\mathbf{D}.4$ to $\mathbf{D}(L_{i-1})$, then $L_{i-1} = g; \bar{G}$, for some expression g and sequence of expressions $\bar{G}$. In $\Delta(\mathbf{D}(e))$, $\mathbf{D}(L_{i-1})$ has two children $\mathbf{N}(g)$ and $\mathbf{D}(L_i)$. By the definition of *target* procedures in Fig. 1 $L_i = \bar{G}; g$. In the children of $\mathbf{D}(L_{i-1})$, both g and $\bar{G}$ are the traces of the corresponding nodes in L_{i-1}. By the induction hypothesis, every element of L_{i-1} is a subexpression or a state of the computation of e. By the definition of **N** in Fig. 1, the execution of $\mathbf{N}(g)$ results in the application of rewrite and/or pull-tab steps to g. Hence, the trace of every element in L_{i-1} is a state of the computation of some subexpression of e, and consequently every element in L_i is a subexpression or a state of the computation of e. □

Corollary 2 (Simulation). *Let S be a* source *program,* **D** *the dispatch procedure of the corresponding* target *program, and e an expression of S. $\omega(\mathbf{D}(e))$ is a pull-tabbing derivation of e.*

Proof. By the definition of the *target* procedures in Fig. 1, every step in $\omega(\mathbf{D}(e))$ is a rewrite or pull-tab step of some expression which, by Lemma 6 is a subexpression of a state of a computation of e. Thus, in the context of e, the sequence of these steps is a pull-tabbing derivation of e. □

Corollary 2 shows that a computation in the *target* program can be seen as a pull-tabbing computation in the *source* program. Each element in the queue argument of **D** is a subexpression s of a state of a computation t of the top-level expression e. Expression t is not explicitly represented. Every node in the path from the root of t to s, excluding the root of s, is labeled by the *choice* symbol. Hence, any value of s is a value of e. Furthermore, s can be evaluated independently of any other element of the queue argument of **D**, though it may share subexpressions with them, which both improves efficiency and simplifies computing the values of e.

As presented in Fig. 1, the queue argument of **D** may contain expressions that combine the left and right alternatives of the same *choice*, an example of which is in Fig. 2. These expressions are unintended. The following statement characterizes all and only the intended values. A simple modification of **D**, discussed shortly, avoids creating these unintended expressions in the *target* program. A *consistent computation*, formally defined in [7, Def. 4], avoids combining the left and right alternatives of the clones of a same *choice* produced by pull-tab steps.

Theorem 2 (Correctness). *Let S be a* source *program,* **D** *the dispatch procedure of the corresponding* target *program, e an expression of S, and $\omega(\mathbf{D}(e)) = t_0 \Rrightarrow t_1 \Rrightarrow \ldots$ the simulated computation of e. Modulo a renaming of nodes: (1) if $e \xrightarrow{*} v$ in S, for some value v of S, and t_k is an element of $\omega(\mathbf{D}(e))$, for some $k \geqslant 0$, then $t_k \xrightarrow{*} v$, for some consistent computation in S; and (2) if t_k is an element of $\omega(\mathbf{D}(e))$, for some $k \geqslant 0$, and $t_k \xrightarrow{*} v$ is a consistent computation in S, for some value v of S, then $e \xrightarrow{*} v$ in S.*

Proof. By Corollary 2, $e \stackrel{*}{\Rightarrow} t_k$ defines a pull-tabbing derivation of e in S that executes no *choice* steps [7, Def. 4]. Therefore, points (1) and (2) are direct consequences of (1) and (2), respectively, of [7, Th. 1]. □

Given an expression e of the *source* program, we evaluate $\mathbf{D}(e)$ in the *target* program. From any state of the computation of e, through consistent computations, we find all and only the values of e in S. Point (1) ensures a weak form of *completeness*—from any state of the computation of e in *target* program it is possible to produce any value of e. Point (2) ensures the *soundness* of the *fair scheme*—the *target* program does not produce any value of e that would not be produced in the *source* program. We will address the weakness of our completeness statement shortly.

The consistent computations sought for obtaining the values of e come almost for free with the *fair scheme*. A simple modification of **D** eliminates inconsistencies so that only intended values are produced. A *fingerprint* [20] is a finite set $\{(c_1, a_1), \ldots, (c_j, a_j)\}$, where c_i is a *choice* identifier [7, Def. 1] and $a_i \in \{1, 2\}$. A fingerprint is associated to a path in an expression. Given an expression e and a path $p = n_0, n_1, \ldots$ in e starting at the root of e, the fingerprint of p in e, denoted $F_e(p)$, is defined by induction on the length of p as follows. Base case: $F_e(n_0) = \varnothing$. Ind. case: Let $f = F_e(n_0, n_1, \ldots n_k)$, for $k \geqslant 0$. If n_k is labeled by the *choice* symbol and has *choice* identifier i, then $F_e(n_0, n_1, \ldots n_{k+1}) = f \cup \{(i, h)\}$, where $h = 1$, resp. $h = 2$, iff n_{k+1} is the first, resp. second, successor of n_k. Otherwise, n_k is not labeled by the *choice* symbol, and $F_e(n_0, n_1, \ldots n_{k+1}) = f$. A fingerprint f is *inconsistent* iff for some *choice* identifier i both $(i, 1)$ and $(i, 2)$ are in f. Pull-tabbing creates expressions reachable through paths with inconsistent fingerprints, see Fig. 2 for an example. These paths should be ignored.

An implementation associates a fingerprint to each expression in the queue argument of **D**. Expressions with consistent fingerprints are evaluated as discussed earlier whereas expressions with inconsistent fingerprints are removed from the queue

```
D(g; Ḡ) =
  if fingerprint(g) is consistent
    then case g of
      ... rules as in Fig. 1 ...
    else D(Ḡ);
```

Fig. 4. *Refinement of the dispatch procedure to avoid evaluating inconsistent expressions.*

We close this section with a final interesting property of the *fair scheme* unrelated to its correctness or optimality. Let $\bar{G}$ be the argument of **D** during the computation of some expression e, and g_1 and g_2 two elements of $\bar{G}$. If a step is computed on both g_1 and g_2, then the redex (including pull-tabs) patterns [17, Def 2.7.3] of these steps are either disjoint or the same since the only overlapping rules are (1) and these rules are not used. This implies that distinct steps over distinct elements of $\bar{G}$, i.e., distinct needed steps of e, can be executed simultaneously. Thus, the *fair scheme* is able both to extract some parallelism from a program without having the programmer to explicitly

encode or annotate the program for parallel execution, and to exploit this parallelism during the execution of the program.

5 Strong Completeness

The completeness statement of Th. 2 is weak since, e.g., any hypothetical *target* program that keeps rewriting any expression to itself satisfies the same completeness statement. Of course, rewriting any expression to itself is useless, whereas our *target* program rewrites only needed redexes (Th. 1.2).

We believe that if S is a *source* program and T is the corresponding *target* program, for any expression e, if v is a value of e in S, then v is eventually produced by T. In the following, first we argue why a proof of this claim does not appear to be a low-hanging fruit. Then we describe a construction that might lead to a formal proof.

In orthogonal systems, repeatedly reducing needed redexes eventually leads to a normal form, if it exists [34, Th. 3.26]. This seminal result does not cross over to our framework. We work with a particular kind of graph instead of terms, but we believe that this difference is not crucial; actually it simplifies some aspects of the discussion. We adopt a different definition of need, but we believe that this difference is not crucial either, since the two notions coincide on the deterministic portions of a computation. The reason why the theorem of [34] does not extend to *LOIS* systems is the *choice* operation. Consider the rewrite rule:

$$f(n) \to f(n+1)\,?\,n \tag{6}$$

The (infinite) derivation $f(0) \overset{+}{\to} f(1)\,?\,0 \overset{+}{\to} f(2)\,?\,1\,?\,0 \overset{+}{\to} \ldots$ makes only steps without which some value of $f(0)$ could not be reached. Hence, in an intuitive sense these steps are needed for those values. Yet the derivation does not end in a normal form of $f(0)$ (nor does it end at all). The example suggests that we have to extend not only proofs, but also some of their underlying concepts—as we did for the notion of *need* in Def. 5. In particular, pull-tabbing would benefit from a more flexible notion of termination. In fact, $f(0)$ has an infinite number of normal forms which in practice are all produced by a computation of $f(0)$ that does *not* terminate. Instead of proposing new concepts, a task that seems impossible within the confines of this paper, we take a different route.

Let S be a *source* program, T the program obtained from S according to the *Fair Scheme*. Informally speaking, we would like to show that if a computation of e in S produces a value v, then some computation of e in T produces v as well. Formalizing this statement is complicated by the fact that the computation of e in T may not terminate, yet still produce the value v. These considerations suggest to formulate the (strong) completeness of our scheme as follows.

Referring to the previous example, the following derivation shows that $f(0)$ has value 0. Observe that the derivation has not minimal length and, for simplicity, additions steps are omitted.

$$f(0) \overset{+}{\to} f(1)\,?\,0 \overset{+}{\to} f(2)\,?\,1\,?\,0 \overset{+}{\to} 0 \tag{7}$$

Below, we show the same derivation with an explicit representation of nodes labeled by operations. Without loss of generality, let $\{1, 2, 3, \ldots\}$ be the set of nodes. We deviate from the customary linear notation of graphs used in the rest of the paper for a reason that will become clear shortly. Nodes, which are arbitrarily chosen, are shown as subscripts of the symbols labeling them.

$$f_1(0) \to f_2(1)\,?_3\,0 \to f_4(2)\,?_5\,1\,?_3\,0 \to 0 \tag{8}$$

Now, we combine a symbol/node pair into a new symbol, called *subscripted symbol*, and construct from S a new graph rewriting system S' that computes with subscripted symbols. For the running example, the rules of S' are shown below, where again for simplicity addition is not subscripted since it does not play any significant role.

$$\begin{array}{l} f_1(n) \to f_2(n+1)\,?_3\,n \\ f_2(n) \to f_4(n+1)\,?_5\,n \\ x\,?_3\,y \to y \end{array} \tag{9}$$

In S', the rules of a subscripted symbol originating from a non-*choice* symbol of S are the same as in S except for the presence of subscripts. A symbol of S might generate a family of symbols of S' that differ only for the subscripts, see the first two rules of (9). A subscripted *choice* has a single rule yielding either the left or right argument according to the rule used in the derivation of $e \xrightarrow{*} v$ in S, see the third rule of (9). Using graph rewriting instead of term rewriting is essential for meaningful subscripting of *choices*. In *term* rewriting, a symbol may have two residuals [34], i.e., be duplicated, by a step. If a *choice* is duplicated, one occurrence could be reduced to the left argument whereas the other occurrence to the right argument. This would prevent defining a subscripted *choice* with a single rule. In *graph* rewriting, every node has at most one residual (is never duplicated) by a step. Hence a subscripted *choice* is reduced at most once and our construction of S' is sensible.

System S' is inductively sequential since every operation has a definitional tree. In particular, for any non-*choice* operation f and subscript i, f_i in S' has the same definitional tree(s) as f in S except for the subscripts. The inductive sequentiality of subscripted *choices* is trivial. Thus, S' is orthogonal [32]. The fact that reducing needed redexes is a (hyper)normalizing strategy is proved for orthogonal *term* rewriting systems. We believe that the same proof holds for the expressions (acyclic, single-root graphs in inductively sequential graph rewriting systems) that we consider. Thus, we assume that reducing needed redexes in S' is normalizing.

If we compile S' according to the *Fair Scheme* and obtain a *target* T', then the simulated computation of the subscripted e, which we denote e', will reduce only redexes that by Th. 1 are needed according to our definition and by Lemma 2 are needed according to [34]. Hence, the simulated computation of e' in T' produces *the* normal form of e' in S' which is *a* subscripted normal form of e in S. Since only operation symbols are subscripted, these normal forms are equal (modulo a renaming of nodes).

If we compile S according to the *Fair Scheme* and obtain a *target* T, then the simulated computation of e in T will reduce the same redexes reduced in the computation of e' in T' as follows. Non-*choice* symbols have the same definitional trees, hence non-*choice*-rooted redexes undergo exactly the same steps in both *targets* except for the

subscripts. In T', a *choice*-rooted needed redex is reduced to either of its alternatives. In T, the same (except for subscripts) redex is not (explicitly) reduced. Rather, the *choice* is pulled up in a way that, by Th. 2, allows us to produce each expression that would be obtained by reducing the *choice* to either of its alternatives. Indeed, both these expressions are produced by rule **D**.1 which has the same effect as reducing a *choice* to each of its alternatives. Hence, the normal form of e' computed in T' is computed in T as well. In the above argument, our construction is rigorous, but we assumed that some facts about term rewriting systems cross over to graph rewriting systems of the same class. For this reason, we present the strong completeness of our compilation scheme as a conjecture.

Statement (Strong Completeness) *Let S be a* source *program,* **D** *the dispatch procedure of the corresponding* target *program, and e an expression of S. If $e \xrightarrow{*} v$ in S, for some value v, then $\Delta(\mathbf{D}(e))$ has a node $\mathbf{D}(v; \bar{G})$ for some, possibly empty, sequence of expressions $\bar{G}$.*

The above statement is exactly what we need in practice. If e has value v, node $\mathbf{D}(v; \bar{G})$ of $\Delta(\mathbf{D}(e))$ is where v becomes available for consumption.

6 Implementation

An implementation of the *Fair Scheme* in C++ [36] is under way. Very preliminary benchmarks, that focus on the functional aspects of the implementation's *back end* are presented in [15].

An expression e of the *source* program is represented by a C++ class, *Node*, abstracting the node at the root of e. Specialized subclasses of *Node* are defined for operations, constructors, the *choice*, and failures. Although structurally different, all these subclasses have the same storage size. This constraint enables efficient subexpression replacement. When a redex is replaced, the root node of the replacement is placed in the *same* memory location that was previously allocated for the root node of the redex through a unique feature of C++ called *placement new*. This approach eliminates the need for pointer redirection [23, Def. 8], which saves many small, though cumulatively expensive, operations. Replacement in place by collapsing rules, i.e., rewrite rules whose right-hand side is a variable, is known to potentially duplicate computations. We currently avoid this problem with "indirection nodes" [39, Sec. 8.1].

A node has a handful of attributes including references to its successors. Since the storage size of a node is constant, nodes labeled by symbols of arity three or greater link to overflow storage to accommodate all of the successors. A node attribute called *tag* tells whether the label of a node is an operation, a constructor, a failure, etc. For constructor symbols, the tag also identifies the specific constructor. The tag is an integer in a small range that supports efficient pattern matching by means of a jump table.

The *target* procedures are implemented by C++ methods of class *Node*. For the most part, the implementation is close to the definition presented in Fig. 1 with a few adjustments. The argument of function **D** is a queue of objects consisting of an expression and a fingerprint. The fingerprint is used to discard inconsistent expressions in the manner explained at the end of Sec. 4. The *Fair Scheme* as defined in Fig. 1 executes a

step of an expression e by traversing e from the root of e to the root of some the redex t. Executing a traversal for each step ensures the termination of **N**, which is essential for the strong completeness of the *Fair Scheme*. We have shown in Th. 1 that the step at t is needed for e. Consequently, if the reduct of t is not constructor rooted, another step at (the reduct of) t is needed. This situation is likely to result in repeated traversals from the root of e down to the root, say n, of t. A simple and effective optimization is to keep reducing e at n until either the label of n becomes a constructor symbol or a fixed number of steps has been executed. The latter preserves the completeness of the scheme.

A second major optimization of the *Fair Scheme* concerns rule **D**.3 of Fig. 1. A naive test for determining whether an expression g is a value entails the traversal of g. This traversal is avoided by storing a flag in each node. This flag is set by the implementation of *target* procedure **N** and makes the test of rules **D**.3 fast and straightforward. The flag is also used by rule **N**.3. The recursive application of **N** to a successor of the root is skipped, if the flag reports that the successor is a value already.

7 Related Work

Our work principally relates to the implementation of functional logic languages [19, 21, 22, 24, 30, 45]. This is a long-standing and active area of research whose difficulties originate from the combination of laziness, non-determinism and sharing [41].

The 90's saw various implementations, such as PAKCS [30] and $\mathcal{TOY}$ [22], in which Prolog is the target language. This target environment provides built-in logic variables, hence sharing, and non-determinism through backtracking. The challenge of these approaches is the implementation in Prolog of lazy functional computations [27].

The following decade saw the emergence of virtual machines, e.g., [14, 33, 44, 45], with a focus on operational completeness and/or multithreading. In some very recent implementations [18, 19, 24] Haskell is the target language. This target environment provides lazy functional computations and to some extent sharing. The challenge of these approaches is the implementation of non-determinism in Haskell.

Our approach follows [16], which relies less on the peculiarities of the target environment than most previous approaches. The *target* procedures, being abstract, can be mapped to a variety of programming languages and paradigms. For example, [16] maps to OCaml [46] using its functional, but not its object-oriented, features.

Our work extends the *Basic Scheme* of [16]. The *Fair Scheme* is fair in the sense that any subexpression of a state of a computation which could produce a result is eventually reduced with a needed step. Fairness ensures that, given enough computational resources, all the values of any expression are eventually produced, a very desirable property of computations in any declarative language. We showed that achieving fairness is both conceptually simple, the complexities of the definitions of *Fair* and *Basic Scheme* are comparable, and computationally feasible, preliminary results show that the performances our implementation [15] and [16] are comparable. One major contribution of the *Fair Scheme* is its provability. No proof of optimality is given in [16] and the *Basic Scheme* is not strongly complete.

A strategy for the same class of *source* programs accepted by our compiler is in [3]. This strategy executes rewrite (and narrowing) steps, but not pull-tabs, and is non-deterministic, i.e., it assumes that a *choice* is *always* reduced to the "appropriate" alternative to produce a result, when there exists such a result. This assumption is obviously unrealistic. In practice, all implementations of [3] resolves this non-determinism in one way or another, but without any guarantees. By contrast, the *Fair Scheme* strategy is deterministic and its essential properties are well-understood and provable.

8 Conclusion

We presented the design of a compiler for functional logic programming languages. Our compiler is abstract and general in the sense that both *source* programs input to the compiler and *target* programs output from the compiler are encoded in intermediate languages. This separation greatly contributes to the flexibility of our compilation scheme. A *source* program is a graph rewriting system obtainable from a program in a concrete syntax such as Curry and $\mathcal{TOY}$. A *target* program consists of three procedures that make recursive calls and rewriting and pull-tab steps. From these procedures, it is easy to obtain concrete code in any number of programming languages.

Our compiler is remarkably simple—it is described by the 15 rules presented in Fig. 1. The simplicity of the compiler description enables us to prove properties of the compilation to a degree unprecedented for a work of this kind. We showed both correctness and optimality. Loosely speaking *correctness* means that the *target* code produces all and only the results produced by the *source* code, and *optimality* means that the *target* program makes only steps that the *source* program must make to obtain a result.

The focus of this paper has been formalizing the *Fair Scheme* and discovering and proving some of its fundamental properties. Future work will focus on the implementation [15]. The presentation of the *Fair Scheme* in Fig. 1 is conceptually simple and suitable to prove various properties of the computations of the *target* program. This presentation is not intended as a faithful or complete blueprint of an implementation.

The *Fair Scheme* is the only deterministic strategy for non-deterministic functional logic computations with a proof of optimality and correctness.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proceedings of the Third International Conference on Algebraic and Logic Programming*, pages 143–157, Volterra, Italy, September 1992. Springer LNCS 632.
3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proceedings of the Sixth International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30, Southampton, UK, September 1997. Springer LNCS 1298. Extended version at `http://cs.pdx.edu/~antoy/homepage/publications/alp97/full.pdf`.

4. S. Antoy. Constructor-based conditional narrowing. In *Proc. of the 3rd International Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 199–206, Florence, Italy, September 2001. ACM.
5. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
6. S. Antoy. Programming with narrowing. *Journal of Symbolic Computation*, 45(5):501–522, May 2010.
7. S. Antoy. On the correctness of pull-tabbing. *TPLP*, 11(4-5):713–730, 2011.
8. S. Antoy, D. Brown, and S. Chiang. Lazy context cloning for non-deterministic graph rewriting. In *Proc. of the 3rd International Workshop on Term Graph Rewriting, Termgraph'06*, pages 61–70, Vienna, Austria, April 2006.
9. S. Antoy, D. Brown, and S.-H. Chiang. On the correctness of bubbling. In F. Pfenning, editor, *17th International Conference on Rewriting Techniques and Applications*, pages 35–49, Seattle, WA, August 2006. Springer LNCS 4098.
10. S. Antoy and M. Hanus. Functional logic design patterns. In *Proceedings of the Sixth International Symposium on Functional and Logic Programming (FLOPS'02)*, pages 67–87, Aizu, Japan, September 2002. Springer LNCS 2441.
11. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Twenty Second International Conference on Logic Programming*, pages 87–101, Seattle, WA, August 2006. Springer LNCS 4079.
12. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2009)*, pages 73–82, Lisbon, Portugal, September 2009.
13. S. Antoy and M. Hanus. Functional logic programming. *Comm. of the ACM*, 53(4):74–85, April 2010.
14. S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A virtual machine for functional logic computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pages 108–125, Lubeck, Germany, September 2005. Springer LNCS 3474.
15. S. Antoy and A. Jost. A target implementation for high-performance functional programs. In *Presentation at the 14th International Symposium Trends in Functional Programming (TFP 2013)*, Provo, Utah, 2013. Available at `http://web.cecs.pdx.edu/~antoy/homepage/publications/tfp13/paper.pdf`.
16. S. Antoy and A. Peters. Compiling a functional logic language: The basic scheme. In *Proc. of the Eleventh International Symposium on Functional and Logic Programming*, pages 17–31, Kobe, Japan, May 2012. Springer LNCS 7294.
17. M. Bezem, J. W. Klop, and R. de Vrijer (eds.). *Term Rewriting Systems*. Cambridge University Press, 2003.
18. B. Brassel. *Implementing Functional Logic Programs by Translation into Purely Functional Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2011.
19. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
20. B. Brassel and F. Huch. On a tighter integration of functional and logic programming. In *APLAS'07: Proceedings of the 5th Asian conference on Programming languages and systems*, pages 122–138, Berlin, Heidelberg, 2007. Springer-Verlag.
21. B. Brassel and F. Huch. The Kiel Curry System KiCS. In D. Seipel and M. Hanus, editors, *Preproceedings of the 21st Workshop on (Constraint) Logic Programming (WLP 2007)*, pages 215–223, Würzburg, Germany, October 2007. Technical Report 434.
22. R. Caballero and J. Sánchez, editors. *TOY: A Multiparadigm Declarative Language (version 2.3.1)*, 2007. Available at `http://toy.sourceforge.net`.

23. R. Echahed and J. C. Janodet. On constructor-based graph rewriting systems. Technical Report 985-I, IMAG, 1997. Available at ftp://ftp.imag.fr/pub/labo-LEIBNIZ/OLD-archives/PMP/c-graph-rewriting.ps.gz.
24. S. Fischer, O. Kiselyov, and C. Chieh Shan. Purely functional lazy nondeterministic programming. *J. Funct. Program.*, 21(4-5):413–465, 2011.
25. W. Fokkink and J. van de Pol. Simulation as a correct transformation of rewrite systems. In *In Proceedings of 22nd Symposium on Mathematical Foundations of Computer Science, LNCS 1295*, pages 249–258. Springer, 1997.
26. J. C. González Moreno, F. J. López Fraguas, M. T. Hortalá González, and M. Rodríguez Artalejo. An approach to declarative programming based on a rewriting logic. *The Journal of Logic Programming*, 40:47–87, 1999.
27. M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *LOPSTR '95: Proceedings of the 5th International Workshop on Logic Programming Synthesis and Transformation*, pages 252–266, London, UK, 1996. Springer-Verlag.
28. M. Hanus, editor. *Curry: An Integrated Functional Logic Language (Vers. 0.8.2)*, 2006. Available at http://www-ps.informatik.uni-kiel.de/currywiki/.
29. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
30. M. Hanus, editor. *PAKCS 1.9.1: The Portland Aachen Kiel Curry System*, 2008. Available at http://www.informatik.uni-kiel.de/~pakcs.
31. M. Hanus, H. Kuchen, and J. J. Moreno-Navarro. Curry: A truly functional logic language. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, Portland, Oregon, 1995.
32. M. Hanus, S. Lucas, and A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8, 1998.
33. M. Hanus and R. Sadre. An abstract machine for Curry and its concurrent implementation in Java. *Journal of Functional and Logic Programming*, 1999(Special Issue 1):1–45, 1999.
34. G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*. MIT Press, Cambridge, MA, 1991.
35. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
36. ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012.
37. T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 190–203, Nancy, France, 1985. Springer-Verlag, LNCS 201.
38. J. F. T. Kamperman and H. R. Walters. Simulating TRSs by minimal TRSs a simple, efficient, and correct compilation technique. Technical Report CS-R9605, CWI, 1996.
39. J. R. Kennaway, J. K. Klop, M. R. Sleep, and F. J. de Vries. The adequacy of term graph rewriting for simulating term rewriting. In M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors, *Term Graph Rewriting Theory and Practice*, pages 157–169. J. Wiley & Sons, Chichester, UK, 1993.
40. F. J. López-Fraguas and J. de Dios-Castro. Extra variables can be eliminated from functional logic programs. *Electron. Notes Theor. Comput. Sci.*, 188:3–19, 2007.
41. F. J. López-Fraguas, E. Martin-Martin, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and narrowing for constructor systems with call-time choice semantics. *Theory and Practice of Logic Programming*, pages 1–49, 2012.
42. F. J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN*

international conference on Principles and practice of declarative programming, pages 197–208, New York, NY, USA, 2007. ACM.

43. F. J. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proceedings of the Tenth International Conference on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
44. W. Lux. An abstract machine for the efficient implementation of Curry. In H. Kuchen, editor, *Workshop on Functional and Logic Programming*, Arbeitsbericht No. 63. Institut für Wirtschaftsinformatik, Universität Münster, 1998.
45. W. Lux, editor. *The Münster Curry Compiler*, 2012. Available at `http://danae.uni-muenster.de/~lux/curry/`.
46. Ocaml. Available at `http://caml.inria.fr/ocaml/index.en.html`, 2004.
47. D. Plump. Term graph rewriting. In H.-J. Kreowski H. Ehrig, G. Engels and G. Rozenberg, editors, *Handbook of Graph Grammars*, volume 2, pages 3–61. World Scientific, 1999.
48. D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.