Compiling a Functional Logic Language:
The Basic Scheme

Sergio Antoy Arthur Peters

Computer Science Dept., Portland State University, Oregon, U.S.A.
{antoy,amp4}@cs.pdx.edu

Abstract. We present the design of a compiler for a functional logic programming language and discuss the compiler's implementation. The *source* program is abstracted by a constructor based graph rewriting system obtained from a functional logic program after syntax desugaring, lambda lifting and similar transformations provided by a compiler's front-end. This system is non-deterministic and requires a specialized normalization strategy. The *target* program consists of 3 procedures that execute graph replacements originating from either rewrite or pull-tab steps. These procedures are deterministic and easy to encode in an ordinary programming language. We describe the generation of the 3 procedures, discuss the correctness of our approach, highlight some key elements of an implementation, and benchmark the performance of a proof-of-concept. Our compilation scheme is elegant and simple enough to be presented in one page.

1 Introduction

Our goal is the efficient execution of functional logic computations for the implementation of programming languages such as Curry [25] and $\mathcal{TOY}$ [16]. A functional logic language offers functional application, as found in Haskell, ML and Scheme, and logic (also called free or unbound) variables, as found in Prolog. The logic variables introduce non-determinism. Functional logic languages also offer a second, more function-oriented, form of non-determinism, "non-deterministic functions". A non-deterministic function (some people prefer to call it "operation") is a function-like symbol that when applied to some argument returns one among several results. Logic variables and non-deterministic functions, although apparently very different, are equivalent [7,33] in the sense that one is easily replaceable by the other without changing a program's meaning. Current functional logic languages provide syntax for both forms of non-determinism for the convenience of the programmer.

Non-determinism is frequently and conveniently used in programming when the information to make "the right choice" is missing or incomplete. For example, consider a program for solving the n-queens puzzle. The program places one queen after another on the board, but the information for appropriately choosing rows and columns of a placement is incomplete. Thus, rows and columns are non-deterministically chosen and each choice is constrained to ensure the solution of the puzzle. In many situations, constraining a value that solves a problem is much simpler than computing that value [5]—even when the information for computing that value is available.

The Eleventh International Symposium on Functional and Logic Programming
T. Schrijvers and P. Thiemann (Eds.): (FLOPS 2012), LNCS 7294, pp. 17–31, 2012.
Kobe University, Kobe, Japan, May 23–25, 2012
The original publication is available at http://www.springerlink.com

While non-determinism in a functional setting is very expressive and convenient for the programmer [5], its implementation, particularly in combination with laziness and sharing, is difficult. This paper presents a relatively simple, complete and compact solution to this problem.

2 The Basic Scheme

In this section we formalize the programs to which our design is applicable and the compilation scheme, which we call the *basic scheme*, for these programs. We assume some familiarity with the concepts of functional logic programming [9,21,22,24,26] and graph rewriting [17,18,36] as a formal model of functional logic computations.

2.1 Symbols and Expressions

A *program* is a pair $(\Sigma \cup \mathcal{X}, \mathcal{R})$ in which $\Sigma = \mathcal{C} \uplus \mathcal{D}$ is a *signature* partitioned into *constructors* and *operations* (or functions), $\mathcal{X}$ is a denumerable set of (bound) *variables*, and $\mathcal{R}$ is a set of *rewrite rules* with the characteristics discussed below. Without further mention, we assume that the signature is many sorted and that any expression, to be defined shortly, over the signature is well typed. Each rule's left-hand side is a *pattern*, i.e., an *operation* symbol applied to zero or more expressions consisting of *constructor* symbols and/or variables only. Each operation in $\mathcal{D}$ is *inductively sequential* [1], i.e., its rewrite rules are organized in a hierarchical structure called a *definitional tree* whose definition is given in the next section.

Non-determinism is abstracted by a binary, infix, polymorphic operation, denoted "**?**" and called the *choice* operator, and defined by the rules:

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

We will never apply the choice's rules in a computation for reasons that will be presented shortly.

Each occurrence of the choice symbol is tagged with an identifier [6] which is not a part of the source program. This identifier is used during *pull-tab* steps which are executed by our compilation scheme and will be defined shortly. The identifier of a choice is denoted as a subscript of the choice symbol. We make the convention that every time a node n labeled by the choice symbol is created either for a top-level expression or by a rewrite, the choice identifier of n is fresh.

A *term graph*, also called an *expression*, is defined in the customary way [17, Def. 2], but we extend the decorations of some nodes with a choice identifier [6, Def. 1]. An expression e is a *value* iff 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., **`1/0`** and **`head []`**. In a constructor-based system, such expressions are regarded as *failures* or *exceptions* rather than results of computations. The following definition is motivated by our decision of not applying the rules of "**?**".

Definition 1 (Non-deterministic value). *We call an expression e a* non-deterministic value *iff either e is a value or $e = u$ **?** v for some non-deterministic values u and v.*

We may say "deterministic value" *to emphasize that some non-deterministic value is a value.*

2.2 Definitional Trees

A definitional tree is a structure derived from the rewrite rules defining an operation in a program. Our presentation is identical to [1] except for a slightly updated terminology. In particular, the expressions of the definition are graphs [17]. A simple algorithm for constructing definitional trees from the rules defining an operation is in [4]. 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 [17, Def. 5] and $t[p \leftarrow u]$ is the *replacement* by u of the subexpression of t rooted by p [17, Def. 9].

Definition 2. *$\mathcal{T}$ 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 3. *$\mathcal{T}$ 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 4. *We call an operation f of a rewrite system $\mathcal{R}$* 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 $\mathcal{R}$. We call a rewrite system $\mathcal{R}$* inductively sequential *if and only if all operations of $\mathcal{R}$ are inductively sequential.*

Exempt nodes are present in a tree of an incompletely defined operation only. Patterns do not need explicit representation in a definitional tree. However, we will keep them around when their presence simplifies the presentation of our ideas.

2.3 Programs

The programs that we intend to compile are abstracted by a well-studied class of systems, the *limited overlapping, inductively sequential, graphs rewriting systems* (*LOIS*). A general treatment of graph rewriting suitable for our purposes is in [17]. *LOIS* systems are discussed in [4]. In particular, in *LOIS* systems, there is a single operation whose rules' left-hand sides overlap. This is the *choice* operation defined in (1). Source programs are coded in a functional logic language such as Curry or $\mathcal{TOY}$. After desugaring, lambda lifting, firstification, deconditionalization, etc., we obtain *LOIS* systems.

A *LOIS* system can be seen as a set of definitional trees. In the next section, we show how to compile these trees into an executable program.

LOIS systems are an ideal core language for functional logic programs. *LOIS* systems are general enough to perform any functional logic computation [3] and powerful enough to compute by simple rewriting [7] without wasting steps [2]. Also for every *LOIS* system containing *free* (unbound) variables there is an equivalent system that replaces the *free* variables with non-deterministic operations [7,33]. Hence, as in other similar approaches [15], we exclude free variables from our core language. Section 5 will address this point in practice.

2.4 Computations

A *computation* (or *derivation*) of an expression e is a finite or infinite sequence $e = e_0 \Rrightarrow e_1 \Rrightarrow \ldots$ such that $e_i \Rrightarrow e_{i+1}$ is a step. A *step* is a pair of expressions $t \Rrightarrow u$, such that u is obtained from t by either of two transformations, a *rewrite* [17], denoted "$\rightarrow$", or a *pull-tab* [6], denoted "$\Rrightarrow$". The initial expression of a computation is called the *top-level* expression and each element is called a *state of the computation*.

In a Curry or $\mathcal{TOY}$ program, some computations of an expression are "substantially different", i.e., they differ in more than the order of the steps. The outcomes of these computations may include distinct values, exceptions, and/or non-termination. These differences originate from non-deterministic choices, in particular from the application of the choice's rules defined in (1).

Pull-tabbing keeps all the outcomes of an expression in a single structure, the state of the computation. Informally, an application of a symbol s to a choice $x \,\mathbf{?}_i\, y$ is rewritten to $(s\ x)\,\mathbf{?}_i\,(s\ y)$ without committing to either alternative. Pull-tabbing has the useful property that computations of subexpression are automatically shared between alternatives and still evaluated lazily. Any deterministic value of the computation can be "extracted" from the state of the computation. This is in contrast with other approaches to non-deterministic steps, which either select *only one* alternative, e.g., *backtracking*, or manage *multiple* computations, e.g., *cloning*.

2.5 Strategy

An evaluation strategy determines the steps of a computation. Interesting *rewriting* strategies are well understood for several practical classes of functional logic programs [4]. Strategies for non-deterministic computations are typically non-deterministic as well. While this simplifies the formulation of a difficult problem, it leaves to the implementation the burden of selecting which step to execute when the strategy computes many non-deterministic steps. There are cases [28,34] in which this selection sacrifices the strategy's operational completeness. A major contribution of our work is a *deterministic* strategy for non-deterministic computations.

Strategies for computations that include pull-tab steps are scarce and their properties are only partially known. In particular, we are not aware of any result concerning the theoretical performance of any such strategy. Braßel [11] considers a language with *let* and *case* expressions which serves the same purpose as our *LOIS* programs. He proves the soundness and completeness of computations within this language with respect to a

natural semantics based on small steps over *heaps* and *configurations*. Our results are comparable.

2.6 Compilation

We describe the compilation of functional logic programs abstractly. The input of the compilation is a *LOIS* system S, and called the *source* program. We construct the definitional tree of every operation of S's signature except the choice operation, since its rules are not applied. We compile both the signature and the set of these trees of S into 3 *target* procedures denoted **N** (Normalize), **H** (Head-normalize) and **A** (Adjust). At the conceptual level, these procedures are the *target* program, the executable code resulting from the compilation of S.

To present the *target* program, we introduce the notion of trace of a node. This notion allows us to keep track of a subexpression in a graph after the graph undergoes a sequence of replacements.

Definition 5 (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* [17, 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 6 (*Target* procedures). *Each* procedure *of the* target *system takes a graph as argument. Each procedure is defined by cases on its argument. Each case, called a* rule*, is selected by 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* [17, Def. 9] *resulting from either a rewrite* [17, Def. 23]*, or a pull-tab step* [6, Def. 2]*. 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 *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 the empty sequence of actions. The notation $\mathcal{P}_g(d, s)$ is the pull-tab transformation with source s and destination d in g. The notation $g[h \leftarrow e]$ is the replacement in g of h with e. Graphs are written in* linear notation [17, Def. 4]*, e.g., in $g : e$, g is the root node of expression e, with the convention that nodes are explicitly written only when they need to be referenced.*

Procedure **N** computes the values, if any, of an expression of the *source* program. A *representation* in the sense of [6, Def. 5] of these values is obtained by applying **N** to a top-level expression e. Typically, **N** will make recursive calls and/or invoke the

$\mathbf{N}(\mathbf{?}_i\ (n_x : \_, n_y : \_)) = \mathbf{N}(n_x); \mathbf{N}(n_y);$ **N**.1
$\mathbf{N}(g : c(n_{x_1} : \_, \ldots n_{x_k} : \_)) = \mathbf{N}(n_{x_1}); \ldots \mathbf{N}(n_{x_k}); \mathbf{A}(g);$ **N**.2
$\mathbf{N}(g : f(\_, \ldots \_)) = \mathbf{H}(g); \mathbf{N}(g);$ **N**.3

$\mathbf{A}(g : c(p : \mathbf{?}_i\ (\_, \_), \_, \ldots \_)) = \mathcal{P}_g(g, p); \mathbf{A}(L(g)); \mathbf{A}(R(g));$ **A**.1
$\mathbf{A}(g : c(\_, p : \mathbf{?}_i\ (\_, \_), \ldots \_)) = \mathcal{P}_g(g, p); \mathbf{A}(L(g)); \mathbf{A}(R(g));$ **A**.1
$\vdots$
$\mathbf{A}(g : c(\_, \_, \ldots p : \mathbf{?}_i\ (\_, \_))) = \mathcal{P}_g(g, p); \mathbf{A}(L(g)); \mathbf{A}(R(g));$ **A**.1
$\mathbf{A}(c(\_, \_, \ldots \_)) = null$ **A**.2

---

compile $\mathcal{T}$
  case $\mathcal{T}$ **of**
  when $branch(\pi, o, \bar{\mathcal{T}})$ **then**
    $\forall \mathcal{T}_i \in \bar{\mathcal{T}}$ **compile** $\mathcal{T}_i$
    output $\mathbf{H}(g : \pi[o \leftarrow p : \mathbf{?}_i\ (\_, \_)]) = \mathcal{P}_g(g, p);$ **H**.1
    output $\mathbf{H}(g : \pi) = \mathbf{H}(\pi|_o);$ **H**.2
  when $rule(\pi, l \to r)$ **then**
    output $\mathbf{H}(g : l) = g[g \leftarrow r];$ **H**.3
$\mathbf{H}(c(\_, \ldots \_)) = null$ **H**.4

Fig. 1: Compilation of a *source* program with signature Σ into a *target* program consisting of 3 procedures: **N**, **H** and **A**. The rules of **N** and **A** depend only on Σ. The rules of **H** are obtained from the definitional tree of each operation 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. 6. The symbols c and f stand for a generic constructor and operation of the *source* program and i is a choice identifier. A symbol of arity k is always applied to k arguments. L and R denote the left and right successors, respectively, of a choice node. 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.

procedures **H** and **A**. If $\mathbf{N}(e)$ derives e to a non-deterministic value v, then some further processing is necessary to obtain the deterministic values represented by v.

Procedure **A** extracts the deterministic values, if any, produced by a call to **N**. This is obtained by pulling choices higher in an expression until they either reach the root or are just below choices only. In rules labeled **A**.1, any expression that applies a constructor symbol to a choice results in a new choice of two expressions, one for each alternative of the original choice (all the rules except the last one). This transformation brings choices at the top of an expression and obtains alternatives that are choice free. In the rule labeled **A**.2 (the last one), the argument has no choice to pull up, and no action is performed.

Procedure **H** executes rewrite and pulltab steps. A redex of either kind of steps is always operation-rooted. Each operation f of the *source* program contributes a handful of rules defining **H**. We call them $\mathbf{H}_f$*–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 are translated is irrelevant. However, the order among the $\mathbf{H}_f$–rules is relevant. More specific rules are generated first and hence have higher priority. All the $\mathbf{H}_f$–rules

are generated by an abstract procedure, **`compile`**, that traverses a definitional tree, $\mathcal{T}$, of f in post-order. Upon visiting a node of $\mathcal{T}$, **`compile`** generates some rules depending on the node's kind, i.e., *branch*, *rule* or *exempt*. Since there can be several *branch* and *rule* nodes in a definitional tree of operation f, there can be several distinct rules of the same type among the $\mathbf{H}_f$–rules. The last rule, labeled **H**.4, handles situations in which **H** is applied to an expression which is already constructor-rooted. This application will occur only to nodes that are reachable along multiple distinct paths.

Definition 7 (*Target* computation). *Let S be a* LOIS *program and T the* target *program obtained from S with the basic scheme. If A is an action, the computation of A, denoted $\Delta(A)$ is inductively defined as follows. If A is a graph replacement, then $\Delta(A) = A$. Otherwise, $A = Y(e)$ for some target procedure Y of T and some expression e of S. If some rule $l = a_1, a_2, \ldots, a_n$, for $n > 0$, (of highest priority) is applicable to $Y(e)$, i.e., $Y(e) = \sigma(l)$ for some match σ, then $\Delta(Y(e)) = (Y(e), B)$, where $B = \Delta(\sigma(a_1)), \Delta(\sigma(a_2)), \ldots, \Delta(\sigma(a_n))$. Otherwise $\Delta(Y(e)) = Y(e)$. If $\Delta(Y(e))$ is finite, then a left-to-right traversal of its rewrite and pull-tab steps is called the* simulated *computation of e and denoted $\omega(Y(e))$.*

A computation in the *target* program is a possibly infinite, finitely branching, ordered tree in which a branch is an application of a *target* procedure that has a matching rule, whereas a leaf is an application that has no matching rule or either a rewrite or a pull-tab step in the *source* program. Under appropriate conditions, a left-to-right traversal of the computation of $\mathbf{N}(e)$, where e is an expression of the *source* program, visits the sequence of steps of a computation of e in the *source* program.

2.7 Optimization

A rewrite step computed by function **H** is applied to an operation-rooted redex, say t. If this step is $t \rightarrow s$ and s is again operation-rooted, then the basic scheme will again apply function **H** to s in an attempt to derive s to a constructor-rooted expression. This property suggests an optimization which is nearly always very effective. Instead of executing a single step at t, execute an entire derivation starting with t and ending with a non-operation-rooted expression. The implementation whose benchmarks are presented later includes this optimization.

3 Pull-tabbing

The basic scheme implements computations that execute rewrite and pull-tab steps, but never reduce a choice. The idea behind pull-tabbing was originally presented in [15] and further refined in [11]. A detailed description of pull-tabbing and a proof of its correctness in the framework of graph rewriting are in [6]. Below we give an informal account of the intended use of pull-tabbing within the context of our work. During the computation of an expression e, choices are pulled toward the root of the state of the computation. A choice with several predecessors is pulled up toward the root along several paths, and hence the choice is cloned. Each clone of the choice has the same identifier as the original. A choice is never pulled above another choice. The result is a *non-deterministic value* (see Def. 1).

The deterministic values of e are found by traversing the choices at the top of a state of the computation of e. If the left alternative of a choice identified by some i is traversed, then the left alternative of any other choice identified by i must be traversed as well, and likewise for the right alternative. A traversal violating this condition combines together subexpressions originating from mutually exclusive alternatives of the same choice. Nodes and paths on such traversals are called *inconsistent* and must be discarded, since the values that they produce may be unsound. Fig. 2 demonstrates pull-tabbing.

3.1 An example

Let $t =$ `(not x, not x) where x = True ? False`. We evaluate t with the basic scheme. The *source* program defines only the Boolean negation, `not`. The $\mathbf{H}_{\texttt{not}}$-rules are shown below:

$$\begin{array}{l}\mathbf{H}(g : \texttt{not True}) = g[g \leftarrow \texttt{False}];\\ \mathbf{H}(g : \texttt{not False}) = g[g \leftarrow \texttt{True}];\\ \mathbf{H}(g : \texttt{not(}\,?_i\,\texttt{(}h_x : \_\texttt{,}\, h_y : \_\texttt{))}) = g[g \leftarrow ?_i\,\texttt{(not}\,h_x\texttt{,}\ \texttt{not}\,h_y\texttt{)}];\\ \mathbf{H}(g : \texttt{not}\ \ h : \_) = \mathbf{H}(h);\end{array} \tag{3}$$

Informally, the first rule replaces `not True` with `False`. The second rule is similar. Both rules execute a rewrite. The third rule replaces `not(x` $?_i$ `y)` with `(not x)` $?_i$ `(not y)`. This rule executes a pull-tab which "distributes" the application of `not` to the choice's alternatives for further evaluation. The fourth rule is fired only when the argument of `not` is operation-rooted. The argument must be head-normalized in order to head-normalize g.

A compact representation of the evaluation of t by the *target* program is shown in Fig. 2. Each snapshots depicts a state of the computation with applications of *target* procedures to some of its nodes. The third snapshot shows the necessity of tracing. First, procedure **H** is applied a node, say n, labeled by `not`. Then, procedure **N** is applied to the *trace* of n, i.e., the result of the previous application of **H**.

In the last graph of Fig. 2, the applications of **A** do not result in any replacement. As discussed at the beginning of this section, the values of e are found by traversing the choices at the top of the state of the computation. In this example, all the choices have the same identifier and thus are intended as the same choice. There are four traversals, but two of them are discarded because they combine mutually exclusive alternatives of choices with the same identifier. The discarded traversals yield `(True,False)` and `(False,True)` that are not values of t. Thus, the computed values of t are `(True,True)` and `(False,False)`.

4 Correctness

We compile a *source* program S into a *target* program T. The intent is to use T for the computations of S. The advantage is that T defines both which redex to reduce and when to reduce it, while S does neither. Informally speaking, T is S with both a *strategy* and explicit *pull-tab* steps. The latter is quite convenient because neither do we have to irrevocably choose one alternative of a choice over the other alternative nor do we have

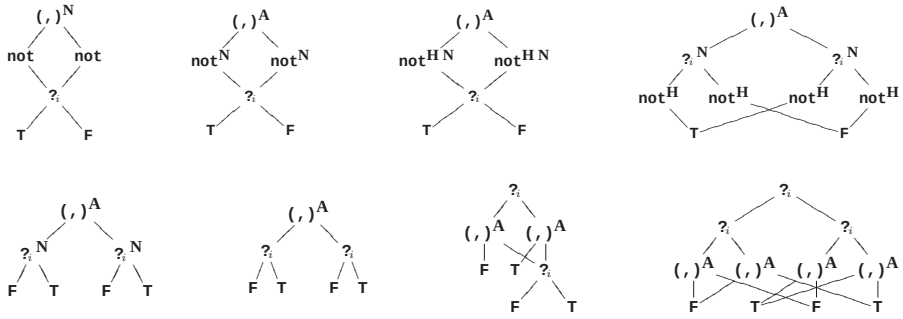

Fig. 2: *Annotated states of the computation of* **`(not x,not x) where x=True?False`**. *The symbols* **`T`** *and* **`F`** *are shorthands for* **`True`** *and* **`False`** *respectively. A superscript of a symbol denotes the application of a* target *procedure to the node labeled by that symbol. The states are in chronological order, but zero or more replacements may be executed between adjacent states.*

to manage multiple computations. Obviously, we expect T to produce the same results that would be produced by S. While at the conceptual level this is true, the statement of correctness is not so direct because T is deterministic whereas S is not. In particular, the *single* computation of $\mathbf{N}(e)$ in T, for some e in S, simulates *all* the computations of e in S and consequently if e has both terminating and non-terminating computations, the computation of $\mathbf{N}(e)$ is non-terminating.

In the statements of this section, the equality of graphs is implicitly modulo a renaming of nodes, a standard practice in graph rewriting [17, Def. 15], since every node introduced by a replacement is "fresh", see also [6, Princ. 1]. The word "simulation" stems from some similarity of our work with transformations of rewrite systems for compilation purposes [20,31].

Lemma 1 (Simulation). *Let S be a* LOIS *program, T the program obtained from S according to the basic scheme, e an expression of S, and Y a procedure of T. If $\Delta(Y(e))$ is finite, then $\omega(Y(e))$ is a pull-tabbing derivation of e in S, i.e., $e = t_0 \Rrightarrow t_1 \Rrightarrow \ldots t_n$, for some $n \geqslant 0$.*

Informally speaking, Lemma 1 shows that a computation in the *target* program can be seen as a pull-tabbing computation in the *source* program. This is instrumental for the correctness of the basic scheme. A *consistent computation* [6, Def. 4] is a derivation that for each choice identifier consistently selects either the left or the right alternative of any choice with that identifier.

Proposition 1 (Correctness). *Let S be a* LOIS *program, e an expression of S, T the* target *program obtained from S by the basic scheme,* **N** *the* Normalize *procedure of T, and $\omega(\mathbf{N}(e)) = t_0 \Rrightarrow t_1 \Rrightarrow \ldots$ Modulo a renaming of nodes:* (1) *if t_k is an element of $\omega(\mathbf{N}(e))$, 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; and* (2) *if $e \xrightarrow{*} v$ in S, for some value v of S, and t_k is an element of $\omega(\mathbf{N}(e))$, for some $k \geqslant 0$, then $t_k \xrightarrow{*} v$, for some consistent computation in S.*

Given an expression e of the *source* program, we evaluate $\mathbf{N}(e)$ in the *target* program. From any state t of this computation of e, through consistent computations, we find all and only the values of e in S. Point (1) ensures the *soundness* of the basic scheme—the *target* program does not derive any value of e that is not derivable in the *source* program. Point (2) ensures a weak form of *completeness*—from any state of a computation in *target* program it is possible to derive any value of e. The latter is a weak result since, e.g., any hypothetical *target* program that rewrites e to itself ad infinitum satisfies the same completeness claim.

We believe that the basic scheme satisfies a stronger completeness result. If e is an expression of a *source* program S, and T is the *target* program obtained from S with the basic scheme, then every step of the simulated computation of e is needed modulo two appropriate conditions discussed below.

The first condition concerns the fact that pull-tab computations may create subexpressions that are inconsistent in the sense defined earlier. The basic scheme as presented in Fig. 1 ignores this possibility when computing a step. Our implementation passes a *fingerprint* [8,14] to the *target* procedures **N** and **A** and therefore avoids computing steps on subexpressions that are known to be inconsistent.

The second condition concerns the fact—well-know from [2]—that a step computed using definitional trees in *LOIS* systems is needed *modulo a non-deterministic choice*. This condition is perfectly natural when non-determinism is used to abstract lack of information for making "the right choice".

The basic scheme of Fig. 1 suffers from the "left bias". For example, the first rule, **N**.1, attempts to normalize the left alternative of a choice first. If this computation does not terminate, the right alternative will never be considered. Several other rules exhibit the same behavior. The left bias can be avoided by interleaving the evaluation, e.g., one or a few steps at the time, of the left and the right alternative of a choice.

5 Implementation

We implemented the basic scheme in a prototype codenamed *ViaLOIS* consisting of a translator from *source* programs to *target* programs and a small run-time environment. The translator takes as input *FlatCurry* [27], a representation of Curry programs generated by a module of the PAKCS [28] distribution of Curry, and produces as output the 3 *target* procedures encoded in OCaml [35]. The run-time environment provides both support for the execution of the 3 *target* procedures and a few extensions of the basic scheme described below. Our implementation is available at `http://web.cecs.pdx.edu/~amp4/vialois`.

5.1 Representation and replacement

An expression of the *source* program is represented by an OCaml mutable record containing a symbol and the sequence of its arguments. This record abstracts a graph's node, in particular its labeling and successor functions. Symbols come in a handful of variants the most important of which are *constructor*, *operation* and *choice*. *Choice* symbols carry an argument, the choice identifier.

Beside the 3 *target* procedures, the run-time environment provides some functionality for the manipulation of the records representing expressions: accessor functions, printing functions, and most noticeably a procedure for subexpression replacement. Replacements, which originate only from rewrite and pull-tab steps, are "in-place", i.e., through assignments to records representing expressions. This design eliminates the need for pointer redirection [17, Def. 8], which is an expensive operation, but requires that all records have the same structure and that an *indirection node* [32, Sec. 8.1] be used for the replacement of a collapsing rule.

5.2 Extensions

To make the basic scheme practical, the implementation provides the following extensions.

Built-in Types: Built-in types, such as the integers, are allowed in *source* programs. A value of a built-in type is represented by a record whose "symbol" carries a literal, such as an integer. A built-in operation f, such as "$+$" on the integers, is simply the operation's $\mathbf{H}_f$*–rules* hand-coded in OCaml. Adding new built-in types and built-in operations is straightforward.

Variables: Curry allows free variables in source programs' operations' rules. Our formalization excluded these variables. Our implementation represents free variables by a *generator*, a zero-arity function symbol of some type t, which lazily derives to any value of t according to [7]. Variables of large built-in types, such as the integers, are impractical and therefore are not allowed, although [12] shows that variables of type integer can be narrowed if the integers are algebraically defined.

Higher Order: To allow higher-order functions, we introduce two symbols, **`partial`** and **`apply`**, that are not in the *source* program. **`Partial`** acts as a constructor and **`apply`** is a function that manipulates the representation of expressions to handle partial application. This is a standard technique [37] to "firstify" higher-order programs.

Explicit Failure: Earlier we discussed expressions, such as **`head []`**, that cannot be reduced to values (constructor normal forms) because they originate from incompletely defined operations. We represent expressions of this kind with a distinguished symbol called "*fail*". In our compiler, the procedure **`compile`** generates **H**-rules that rewrite to *fail* upon visiting *exempt* nodes. Furthermore, the rules of **H** and **A** are extended to rewrite an expression t to *fail* when an inductive position matches *fail*. For example, the rules in (3) are extended with

$$\mathbf{H}(g : \texttt{not}\ \ \mathit{fail}) = g[g \leftarrow \mathit{fail}]; \tag{4}$$

5.3 Limitations

The translator of *ViaLOIS* does not yet support all *FlatCurry* constructs and some features provided by more mature implementations. However, the supported subset is large

enough to encode any Curry program into an equivalent program that can be translated by *ViaLOIS*. Cyclic expressions are not supported, but recursive values can be converted to nullary functions that build the appropriate infinite value lazily. Modular compilation, functional patterns and set functions are not supported as well, but in our framework, except for the latter, these features entail only modest code extensions that do not affect architecture or the core of our implementation. Hence, their introduction should affect the performance only marginally.

5.4 Performance

Fig. 3 and 4 respectively compare the size in lines of code and the performance on a few benchmarks of several Curry systems. We believe that the small footprints of our compiler and runtime are due only in small part to our implementation's limitations. The implementation of the basic scheme is subjectively very simple, being a straightforward encoding of the rules of Fig. 1, and competitively efficient.

	Compiler	Runtime
ViaLOIS	0.5 (Curry)	0.6 (OCaml)
KiCS2	4.6 (Curry)	1.5 (Haskell)
PAKCS	4.7 (Prolog)	3.3 (Prolog)
MCC	4.3 (Haskell)	9.6 (C)

Fig. 3: Lines of code (in thousands) of several Curry systems. Line counts exclude comments, blank lines, and the standard library. Built-in functions are included as part of the runtime.
- PAKCS is a mature implementation that compiles to Prolog and hence handles non-determinism using backtracking.
- KiCS2 is a recent implementation that compiles to Haskell and uses pull-tabbing for non-determinism.
- MCC is a compiler and virtual machine written in C and based on backtracking.

The benchmarks are:

ChoiceIDs, a non-deterministic benchmark testing the performance of programs with a large number of independent choices. The program non-deterministically generates every integer in a large set looking for a specific value.

PermSort [29], a non-deterministic benchmark testing the performance of non-deterministic search. The program sorts a list of 13 `Int`s using a permutation sort.

Sharing, a non-deterministic benchmark testing for sharing of results between non-deterministic branches. The program performs a permutation sort over a list of 5 numbers computed by a small version of the Tree benchmark.

Tree, a deterministic benchmark testing the performance of data structures and recursion. The program inserts 200,000 pseudo-random numbers into a binary search tree and then counts the number of elements in the tree.

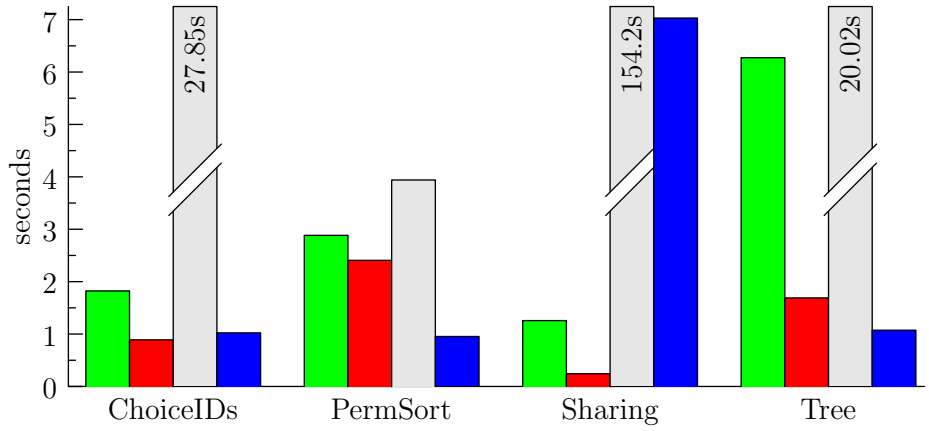

Fig. 4: Benchmark Results. *ViaLOIS*, KiCS2, PAKCS, MCC.

6 Related work and concluding remarks

The implementation of lazy, functional logic languages is a long-standing and active area of research whose difficulties originate from the combination of laziness, non-determinism and sharing.

The 90's saw various implementations, e.g., PAKCS [28], and implementation approaches [23] 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.

The following decade saw the emergence of virtual machines, e.g., [10,30,34], with a focus on operational completeness and/or multithreading. In more recent implementations [11,13,19], 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 relies less on the peculiarities of the target environment than most previous approaches. In fact, in addition to the implementation described in Section 5, we have easily prototyped a different implementation in an object-oriented language in which the nodes of an expression are objects and the target procedures are methods dynamically dispatched on the type of these objects.

The basic scheme is conceptually simple, based on localized graph replacements, and easy to control. Concurrency is a major impulse behind our research and localization of updates, joined with a high degree of control and an independence of any particular run-time environment, makes the basic scheme a good starting point for parallel implementations.

Acknowledgment

We thank Michael Hanus for several interesting discussions on the subject of this paper and for making an early version of KiCS2 available to us.

References

1. 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.
2. 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`.
3. 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.
4. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
5. S. Antoy. Programming with narrowing. *Journal of Symbolic Computation*, 45(5):501–522, May 2010.
6. S. Antoy. On the correctness of pull-tabbing. *TPLP*, 11(4-5):713–730, 2011.
7. 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.
8. 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.
9. S. Antoy and M. Hanus. Functional logic programming. *Comm. of the ACM*, 53(4):74–85, April 2010.
10. 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.
11. B. Brassel. *Implementing Functional Logic Programs by Translation into Purely Functional Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2011.
12. B. Braßel, S. Fischer, and F. Huch. Declaring numbers. *Electron. Notes Theor. Comput. Sci.*, 216:111–124, 2008.
13. 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.
14. 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.
15. 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.
16. R. Caballero and J. Sánchez, editors. *TOY: A Multiparadigm Declarative Language (version 2.3.1)*, 2007. Available at `http://toy.sourceforge.net`.

17. 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.
18. R. Echahed and J. C. Janodet. Admissible graph rewriting and narrowing. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 325 – 340, Manchester, June 1998. MIT Press.
19. S. Fischer, O. Kiselyov, and C. Chieh Shan. Purely functional lazy nondeterministic programming. *J. Funct. Program.*, 21(4-5):413–465, 2011.
20. 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.
21. 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.
22. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
23. 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.
24. M. Hanus. Functional logic programming: From theory to Curry. Technical report, Christian-Albrechts-Universität Kiel, 2005. Available at http://www.informatik.uni-kiel.de/~mh/publications/reports/.
25. M. Hanus, editor. *Curry: An Integrated Functional Logic Language (Vers. 0.8.2)*, 2006. Available at http://www-ps.informatik.uni-kiel.de/currywiki/.
26. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
27. M. Hanus. Flatcurry: An intermediate representation for Curry programs. Available at http://www.informatik.uni-kiel.de/~curry/flat/, 2008.
28. M. Hanus, editor. *PAKCS 1.9.1: The Portland Aachen Kiel Curry System*, 2008. Available at http://www.informatik.uni-kiel.de/~pakcs.
29. M. Hanus. KiCS2 benchmarks. Available at http://www-ps.informatik.uni-kiel.de/kics2/benchmarks/, 2011.
30. 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.
31. 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.
32. 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.
33. 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.
34. 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.
35. Ocaml. Available at http://caml.inria.fr/ocaml/index.en.html, 2004.
36. 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.
37. D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.