Are *Needed* Redexes Really Needed?

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 an approach to rewriting in inductively sequential rewriting systems with a very distinctive feature. In the class of systems that we consider, any reducible term defines a *needed* step, a step that *must* be executed by *any* rewriting computation that produces the term's normal form. We show an implementation of rewriting computations that avoids executing some needed steps. We avoid executing these steps by defining functions that compute a reduct of a step without the explict construction or presence of the redex. Our approach improves the efficiency of many computations—in some cases by one or two orders of magnitude. Our work is motivated by and applicable to the implementation of functional logic programming languages.

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*

General Terms

Languages, Graph, Rewriting, Compilation.

Keywords

Functional Logic Programming Languages, Graph Rewriting Systems, Call-by-Need, Compiler Construction.

1. INTRODUCTION

Functional logic languages such as Curry [16, 18] and $\mathcal{TOY}$ [13, 28] offer a variety of high-level features to the programmer including expressive constructs (e.g., functional patterns, 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).

An approach to compiling these high-level features is to transform a program S into a semantically equivalent graph rewriting system G [11, 14, 34]. This transformation includes lambda lifting [23], elimination of partial applications and high-order functions [39], elimination of conditions [3], replacement of non-inductively sequential functions with inductively sequential ones [3] and replacement of logic (free) variables with generator functions [8, 27]. With this approach [10], a functional logic computation in program S is executed by rewriting some graph according to G.

PPDP '13, September 16–18, 2013, pages 61-71, Madrid, Spain.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2154-9/13/09 ...$15.00.
http://doi.acm.org/10.1145/2505879.2505881.

Graph rewriting is a step-wise process. Given a graph rewriting system G and a graph e over the signature of G, a *step* of e consists of two separate activities: (1) finding a subgraph, t (called the *redex*), of e which is an instance of the left-hand side of a rule $l \to r$ of G, and (2) replacing t in e with the corresponding instance of r (called the *replacement*). A *computation* in G of a graph e is a finite or infinite sequence $e = e_0 \to e_1 \to \ldots$ such that e_{1+1} is obtained from e_i by a step. Each e_i is called a *state* of the computation of e.

An implementation of rewriting repeatedly executes these two activities on some graph representation. In a low-level implementation language, graphs may be represented by dynamic linked structures in which a node is a record (tuple) with components typically abstracting the node's label and successors by means of pointers. The first activity, which we will describe in detail shortly, simultaneously produces a redex, t, a rule, $l \to r$, and a homomorphism, σ such that $t = \sigma(l)$, by looking at the labels of some nodes of e. The replacement, $u = \sigma(r)$, of t is constructed by allocating a new node for each non-variable symbol of r and then setting the node's successors. Replacing t in e can be efficiently executed either by overwriting the components of the root of t with the components of the root of u, or by an indirection pointer.

Recall that in *orthogonal* term rewriting systems [11, Chap. 4], in every reducible term e, there is a redex, called *needed*, which is reduced by *every* computation of e to normal form [20]. In *strongly sequential* systems [21, Sect 4.2], a subclass of the orthogonal systems, this redex is easily found *without lookahead*. *Constructor-based* systems are those in which the signature is partitioned into data *constructors* and defined *operations* [31]. Our results are presented for *inductively sequential* systems [1, 14], which are the intersection [19] of strongly sequential and constructor-based systems.

The systems in which functional logic programs are transformed for execution are called *LOIS* (limited overlapping inductively sequential) [4] and slightly differ from the inductively sequential graph rewriting systems: they rewrite acyclic *graphs* instead of *terms*, and they define an operation called *choice* denoted by the infix symbol "**?**" and defined by the rules:

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

The *choice* operation captures the *logic* component of functional logic programming. Its rules are *overlapping* and applied nondeterministically. Hence *LOIS* systems are not orthogonal. We will postpone any discussion about the *choice* operation to the end of this paper and consider only inductively sequential systems. Throughout this paper we will consider graph rewriting rather than term rewriting, noting differences only where relevant.

For the inductively sequential systems, the redexes and steps of an expression are in a bijection, since each redex is an instance of exactly one rule left-hand side. In graph rewriting, furthermore, a redex has at most one *residual* by a step [20, Def. 2.1]. Hence, in any rewriting computation producing a value, every needed redex must be reduced and it is reduced exactly once. Consequently, a strategy that computes by repeatedly reducing an arbitrarily chosen needed redex is considered optimal [1, 2, 4, 7, 14] in the sense that no other strategy can produce a reduction to the same normal form while executing fewer steps. The intuition seems to suggest that we cannot evaluate an expression *any better*. Our results contradict this intuition and draw some boundaries around the meaning of "needed."

Sect. 2 presents two examples containing familiar functions, which will be used throughout the paper. This section only shows that needed steps can be skipped by some computations that execute more than rewrite steps. Sect. 3 introduces a new notion of need applicable to both redexes and nodes and proves some properties of this notion. This section shows that redexes needed according to the new notion are needed according to the classic notion. Sect. 4 presents two variants of an abstract implementation of computations in inductively sequential rewrite systems and an abstract compiler that generates implementation code. The first variant executes only rewrite steps. The second variant executes both rewrite steps and calls to certain functions, called "ahead fuctions," of the implementation. Sect. 5 discusses how to infer at compile time that some run-time-produced subexpression will be needed. This information is a key component of our implementation. Sects. 6 and 7 address aspects of an implementation with significant potential for improving performance. These aspects are related to the definition and invocation of ahead functions, a key component of our approach. Sect. 8 presents the application of both variants of our implementation to a case study, and compares their relative performances. Sect. 9 briefly outlines the run-time architecture of a concrete implementation of our approach in C++. Sect. 10 describes how to apply the results of our work to the implementation of a functional logic language. Sects. 11 and 12 discuss related work and offer our conclusions, respectively.

2. EXAMPLES

We introduce our approach to rewriting by way of example in the hopes that it will ease the reader's understanding of the formal description, which follows. In this section, we present two examples of evaluations that skip some needed step. Obviously, our computations are not limited to the execution of rewrite steps. Our computing environment or model of computation, which will be defined shortly, is called *ahead*. Both rewrite rules and graphs objects of an *ahead* computation are ordinary. For presentation, we encode them in the syntax of Curry [16], a popular functional logic programming language, and refer to a rewrite system as a "program." Curry's syntax is borrowed from Haskell [33]. Apart from the curried notation for symbol application, infix operations with usual precedence and associativity, and similar syntactic sugar, our programs are very simple and can be directly seen as graph rewriting systems.

An *evaluation* is a finite computation in which the last state is a *constructor form*, i.e., an expression comprising only constructor symbols. Such an expression is also called a *value*. We recall that, in constructor-based systems, a value is a normal form, but some normal forms are not values, e.g. *head* [], where *head* is the usual function that returns the first element of a (non-empty) list. These normal forms are regarded as failures or exceptions. Likewise, a meaningful head-normal form is constructor-rooted and consequently called a *head constructor form*.

Example 1. Consider the following definition of the *absolute value* function:

```
abs n = if 0>n then -n else n                    (2)
```

We use the familiar mixfix notation for the conditional construct in the right-hand side of the rule. In a lazy functional or functional logic language, the conditional construct is only syntactic sugar for the application of an ordinary function of arity three defined by the rules:

```
ifthenelse False x y = y
ifthenelse True  x y = x                         (3)
```

A *rewriting* evaluation of *abs* 2 that makes only needed steps follows. In particular, it is straightforward to verify that the first step is needed, since *abs* 2 is reducible and no other step is available.

$$\begin{aligned}&\mathit{abs}\,2\\ &\quad\to \mathit{ifthenelse}(0>2,-2,2)\\ &\quad\to \mathit{ifthenelse}(\mathit{False},-2,2)\\ &\quad\to 2\end{aligned} \tag{4}$$

Our *ahead* computation follows. Instead of rewriting *abs* 2 according to rule (2), we put the computation on hold. We evaluate $0 > 2$ and upon obtaining *False* we resume the computation on hold, but instead of rewriting *abs* 2 to an instance of the right-hand side of (2), we rewrite it to an instance of the right-hand side of the first rule of (3), i.e., 2. The *ahead* evaluation of *abs* 2 follows, where the nested computation is between braces:

$$\mathit{abs}\,2\ \ \{0>2\to \mathit{False}\}\to 2 \tag{5}$$

By relaxing the rules so that computations may occur by an additional mechanism, some needed steps are avoided. With respect to ordinary rewriting, the *ahead* computation executes one fewer rewrite step and allocates two fewer nodes. It executes the test on *False* that is executed by ordinary rewriting, but in a model of computation where the graph rooted by *ifthenelse* is never created. We could further reduce the work of evaluating *abs* 2 by avoiding the creation of $0 > 2$, and thus skipping another needed step. We will do this later, as the goal of this example is only to show that by slightly changing the laws of computation, a needed step is skipped.

Example 2. Consider the function computing the length of a list:

```
length [] = 0
length (_:xs) = 1 + length xs                    (6)
```

A *rewriting* evaluation of *length* [5,6,7] that makes only needed steps is:

$$\begin{aligned}&\mathit{length}\,[5,6,7]\\ &\quad\to 1+\mathit{length}\,[6,7]\\ &\quad\to \cdots\\ &\quad\to 1+2\\ &\quad\to 3\end{aligned} \tag{7}$$

Our *ahead* computation follows. Instead of rewriting *length* [5,6,7] according to the second rule of (6), we put the computation on hold. We evaluate *length* [6,7] and upon obtaining 2 we resume the computation on hold, but instead of rewriting *length* [5,6,7] to an instance of the right-hand side of the second rule of (6), we rewrite it to 3. The computation executes an addition that is executed by ordinary rewriting, but in a model of computation where the graph rooted by "+" is never created. The *ahead* evaluation of *length* [5,6,7] is:

$$\textit{length}\,[5,6,7]\ \ \{\textit{length}\,[6,7] \overset{*}{\to} 2\} \to 3 \qquad (8)$$

With respect to ordinary rewriting, the *ahead* computation executes one fewer rewrite step and allocates two fewer nodes for each recursive application of *length*. We could further reduce the work of evaluating *length* [5,6,7] by avoiding the creation of *length* [6,7], thus skipping another needed step. We will do this later, as the goal of this example is only to show that by slightly changing the laws of computation, a needed step is skipped. We note in passing that, since this program involves no sharing, it is a term rewriting system, and that for orthogonal term rewriting systems there is a very strong notion of needed redex [20, Def. 3.29].

3. NEED

In this section we introduce a novel notion of "need," we discuss how to compute needed nodes and redexes, and relate this notion to the classic one.

DEFINITION 1 (C-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* c-needed for *e, and similarly is* c-needed for *p, iff in any derivation of e to a head-constructor form, the subexpression of e at n is derived to a head-constructor form. A node n (and the redex rooted by n, if any) of a state e of a computation in S is* c-needed *iff it is c-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* c-needed *iff it reduces only c-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 1-1 correspondence with these 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 c-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 c-needed relation. Finally, our definition is concerned with reaching a *head-constructor* form, not a *normal form*.

Our notion of need generalizes the classic notion [20] with the difference that a needed redex has a replacement, whereas a c-needed node may or may not root a redex. Also, since our systems follow the constructor discipline [31] we are not interested in expressions that do not have a value.

LEMMA 2 (CONSERVATION). *Let S be a* source *program and e an expression of S derivable to a value. If e' is an outermost operation-rooted subexpression of e, and n is both a node c-needed for e' and the root of a redex r, then r is a needed redex of e in the sense of* [20].

PROOF. 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 head-constructor form. By assumption, in any derivation of e' to a head-constructor form r is derived to a head-constructor form, hence it is reduced. Thus, r is a needed redex of e according to [20]. □

By Lemma 2 we drop the prefix "c-" from c-needed.

A definitional tree is a hierarchical organization of the rewrite rules defining certain operations of a program [1, Def. 2] that makes the computation of needed nodes and redexes easy. We recall that if t and u are expressions and p is a node of t, then $t|_p$ is the *subexpression* of t rooted at p [14, Def. 5] and $t[p \leftarrow u]$ is the *replacement* by u of the subexpression of t rooted by p [14, Def. 9].

DEFINITION 3. *$\mathcal{T}$ 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.*

$\mathcal{T} = rule(\pi, l \to r)$, where π is a pattern and $l \to r$ 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 4. *$\mathcal{T}$ 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 5. *An operation f of a rewrite system S 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 S. A rewrite system in which every operation is inductively sequential is referred to as a* source program.

Patterns do not need explicit representation in a definitional tree, but often their presence simplifies the discussion. *Exempt* nodes occur only in trees of incompletely defined operations such as the operation that computes the *head* of a (non-empty) list. The definitional tree of *head* has an exempt node with pattern *head* []. This expression cannot be rewritten and is regarded as a failed computation that does not produce any result.

LEMMA 6 (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, $\sigma(\pi) = e$ for some match σ, 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 [14, 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 $\sigma(o) = p$ for some node p of e. By Def. 3 $\mathcal{T}_1 = \mathcal{T}_0[o_0 \leftarrow c_1(\ldots)]$, 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 7 (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, $\sigma(\pi) = e$ for some match σ, and $\sigma(o) = p$, for some node p of e labeled by an operation symbol, then p is needed for n.*

PROOF. By Lemma 6 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. 3, 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 head-constructor form, l cannot match any state of a computation of e and hence e cannot be reduced at the root. Thus, by Def. 1, p is needed for n. $\square$

4. ABSTRACT IMPLEMENTATION

In this section we discuss an abstract model for implementing graph rewriting. A *program* S is an inductively sequential [1], hence a constructor-based [31], graph rewriting system [14]. An *expression* e of S is an acyclic term graph over the signature of S. The goal is to evaluate e, i.e., to obtain its value according to the rules of S.

The abstract implementation of rewriting is presented as computer code in a simplified form of the programming language Scala [36]. Scala is high-level enough to avoid details that would make the code difficult to present and understand, but low-level enough to retain control over the creation of structures and to measure the computational cost (e.g., in terms of steps and memory allocation) of different variants of the implementation.

We will present two variants of the implementation. The first is an implementation of graph rewriting that reduces only needed redexes. It is simple enough to inspire confidence in its correctness. The second will skip some steps, including needed steps, but will produce the same values as the first variant. This property will maintain our confidence in its correctness.

In our presentation language, we define classes with instance variables and dynamically dispatched virtual methods as in typical object-oriented programming languages. We use pattern matching as provided in Scala. We define static, global functions, which are not directly available in Scala, but are simulated by "companion" objects. We do not declare types, including subtyping, and do not distinguish between constant and mutable values. The concrete language of our implementation is C++. Presenting our ideas in this language would be much harder than in our simplified language due to the lack of certain features, e.g., pattern matching, and amount of details required in by C++ code, e.g., typing, declaring and initializing instance variables, and accessing these variables through explicit manipulation of pointers.

4.1 Variant 1: An Ordinary Implementation

Any symbol f of the signature of a rewrite system is abstracted by a class, identified by f as well, containing a handful of variables and methods. A node n of a graph g labeled by a symbol f is implemented as an instance of class f in which the variables refer to the successors of n. Each class abstracting a signature symbol has a common base class, which we do not show to ease readability. Class f has a method, **H**, intended to execute needed steps of an expression e rooted by a node labeled by f and to derive e to a head-constructor form. Method **H** invoked for a node n derives the expression rooted by n to a head-constructor form, if it exists, and returns it. The body of method **H** of class f is compiled from a definitional tree of operation f. We will present the compiler shortly. By way of example, we present in Fig. 1 the class implementing the operation *length* defined in (6).

```
class length (arg) {
  def H =
    ↓ arg.H match {
      case []      ⇒ new 0
      case _ : xs  ⇒ new +(new 1, new length (xs)).H
    }
}
```

Figure 1: *Implementation of the function computing the length of a list defined in (6). The symbol "↓" denotes the replacement of one graph by another and is described in the text.*

Method **H** is dispatched for an object, referred to as THIS, that provides a context for the execution. THIS abstracts the root node of a graph g labeled by the *length* symbol. In the context of an execution, identifier *arg* abstracts the first and only successor of the root node of g. The Scala keyword **new** identifies a built-in operator that allocates and initializes class instances. The expression **new** $x(y_1, \ldots y_k)$ denotes the construction of a new node labeled by x and with successors $y_1, \ldots y_k$. In particular, **new** 1 stands for a node abstracting integer 1. Allocating a new instance for each step of *length* is not necessary. A single instance of this node suffices for the entire program execution, but we do not want to clutter the presentation with trivial optimizations. Operation **new** *boxes* the integer. We will discuss shortly the boxing and unboxing of built-in types. Reductions are denoted by "↓", are always applied to THIS, which consequently is not explicitly written, and replace THIS. For example, referring to Fig. 1, the expression *length* [] is reduced to a node labeled by the integer 0 as follows: *arg* is reduced to a head-constructor form, i.e., "[]", pattern matched by the first case, and a new node labeled by integer 0 is created and used as replacement. We will discuss later some subtleties concerning replacing a graph by another graph.

Method **H** first attempts to ensure that the variable of THIS, the argument of this instance of *length*, is constructor-rooted. For this purpose, it invokes **H** for *arg*. The root of *arg* might already be labeled by a constructor symbol. Therefore method **H** for any class abstracting a constructor symbol "does nothing." We could avoid the call to **H** in this case with a test, but the test does not appear simpler or more efficient than a call that might have to be executed anyway after the test. After having reduced the variable of this instance of *length* to head-constructor form, a multibranch test selects what to execute depending on the label of the root of *arg*. If the label of the root of *arg* is a constructor, then it must be either "[]" or ":", assuming a well-typed program[1]. In both cases, the appropriate reduction is executed. In the second case, symbol "+" identifies the class abstracting the usual arithmentic addition of integers. If reducing *arg* to head-constructor form is impossible, then the invocation of method **H** for *arg* aborts the computation rather than returning. This behavior is sensible because then THIS cannot be reduced hence derived to head-constructor form.

[1]We use the symbols "[]" and ":" to denote the constructors of empty and non-empty lists of the *functional logic program* as opposed to "[]" and "::" which are the corresponding constructors of class *List* in Scala.

The body of method **H** of class f is produced by compiling the definitional tree of f. Function *compile*, presented below, prints the *body* of method **H**. The function is presented in the same Scala-like language that we use to present the implementation. The construct $\${x}$ in a string, where x is a program element, represents the interpolation of x in the string. We assume a representation of definitional trees in which an inductive variable identifies a subexpression of some expression defined by the context, e.g., an inductive variable is a path in a graph. Within the code of method **H**, the context is THIS. Thus, when interpolated, $\${o}$.**H** is an invocation of **H** for the subexpression of THIS matched by the inductive variable. Function *pattern* takes a (partial) definitional tree and returns the tree's pattern.

```
def compile(𝒯) =
  𝒯 match {
    case rule(_, r) ⇒ print "${r}.H"
    case exempt(_) ⇒ print "abort"
    case branch(_, o, 𝒯′) ⇒ {
      print "{ ${o}.H"
      print "THIS match {"
      for(𝒯″ ← children(𝒯′))
        print "case ${pattern(𝒯″)} ⇒ " compile(𝒯″)
      print "} }"
    }
  }
}
```

Figure 2: *Compiler for the first variant of the implementation. Function compile takes a definitional tree of a functional logic operation f and produces the body of method* **H** *of class f.*

Function *compile* applied to the definitional tree of the operation *length* defined in (6) prints the code fragment of Fig. 3 except that in the figure, we use expressive identifiers, indent lines, and apply operator **new** to each signature symbol (non-variable) occurrence in the right-hand side of a rule. Only the last change is significant, and it is easy to automate, e.g., by a function that takes an expression and checks whether the label of a node is either a variable or a signature symbol. We do not perform this operation because we are going to perform a similar, but more sophisticated, operation later.

```
{ arg.H
  THIS match {
    case length ([])     ⇒ new 0.H
    case length (_ : xs) ⇒ new +(new 1, new length (xs)).H
  } }
```

Figure 3: *Code generated by the compiler for the body of method* **H** *of class* length*, see Fig. 1, except for minor liberties discussed in the text.*

The code of Fig. 3 is functionally equivalent to the body of method **H** of class *length* in Fig. 1. It invokes **H** for 0, which is useless, but harmless. And it pattern matches starting at the root of THIS. The code of Fig. 1 simplifies or optimizes the patterns of the multibranch test by starting at the root of *arg*. Consequently we have "[]" instead of *length* ([]), etc. Adding details to the compiler would eliminate these differences. We do not show these details to ease understanding without unnecessary distractions.

The correctness of the implementation just described is stated as follows. Let e_0 be an operation-rooted needed expression, The execution of **H** for (the root of) e_0 computes a derivation $A = e_0 \to e_1 \to e_2 \cdots$ which is either infinite or finite. When it is finite, the execution either aborts or terminates normally. Regardless of the outcome, every step of this computation is needed. In the first two cases (infinite computation and aborted computation), the need of each step is vacuous. In the third case (terminating computation), the proof is by induction on the length of the derivation. The details of the proof are laborious and rely on the following property: the execution of **H** for (the root of) e_1 computes the same steps as the portion of A starting at e_1, i.e., $e_1 \to e_2 \cdots$.

Only a bit of additional control suffices to evaluate an expression e: repeatedly call **H** on the root of any maximal (outermost) operation-rooted subexpression, g, of any state of a computations of e. The correctness of this approach stems from the fact that any such g must be derived to a head-constructor form to evaluate e. Method **H** is invoked on the root of g. This invocation of **H** aborts the computation if no step of g is available, hence g has no value, hence e has no value. Otherwise it executes only needed steps. A strategy that executes only needed steps is normalizing for orthogonal *term* rewriting systems [20]. We are not aware of a published proof of this claim for *graph* rewriting systems, but the following argument informally proves the claim.

Let S be a program, g an expression of S. Suppose that there exists an infinite computation of g in S, i.e., $A : g \to g_1 \to g_2 \cdots$ that reduces only needed redexes. Remember that we consider acyclic graphs and let $\mathcal{U}$ denote the complete tree unraveling transformation [11, Def. 13.2.9]. By unraveling graphs in rules and computations, we obtain an orthogonal *term* rewriting system $\mathcal{U}(P)$ and a computation of $\mathcal{U}(g)$ such that for every step $t \to s$ of A there is a derivation $\mathcal{U}(t) \xrightarrow{+} \mathcal{U}(s)$ that executes at least one needed step. This implies that $\mathcal{U}(g)$ has no normal form in $\mathcal{U}(P)$, hence g has no value in S.

4.2 Variant 2: An Optimized Implementation

We now introduce the second variant of the implementation. To understand how this variant computes the same values as the first one, although it skips some needed steps, consider the second case of the multibranch test of Fig. 1. First, the reduct is constructed. Method **H** is invoked for the reduct's root, which is labeled by "+". This method is a bit special, since the integers are not algebraically defined, but this peculiarity is largely irrelevant to the point we are making. Class "+" has two variables which stand for the operands or the addition. Method **H** of class "+" first attempts to derive each instance's variable to head-constructor form (a literal integer). Then, it replaces this instance of class "+" with the sum of its variables. The crucial difference with variant 1 is that instead of constructing the reduct, i.e., an instance of class "+", and reducing it, we define a function, denoted by "+" as well, that takes as arguments the variables of the instance of "+" on which **H** operates. Invoking this function produces, if it exists, the same head-constructor form that would be obtained from the reduct that we did not construct.

In the same way, we define a *function* called *length* corresponding to method **H** of *class length*. These functions, which are associated with operations of the signature, are called *ahead functions*. The definition of ahead function *length* is shown in Fig. 4. This function invokes ahead function "+" discussed in the previous paragraph and invokes itself recursively. Observe that in the second case of the multibranch test, operator **new** is not applied to either "+" or *length* because these are function identifiers and we are encoding calls to functions, whereas in method **H** of class *length* they are class identifiers and we are encoding instantiation of classes.

```
def length (arg) =
  arg.H match {
    case [ ]      ⇒  new 0
    case _ : xs   ⇒  +(new 1, length (xs))
  }
```

Figure 4: *Definition of ahead function* length *associated to operation* length*. The code is very similar to method* **H** *of class* length*, see Fig. 1.*

The code in Fig. 4 replaces constructing some nodes with invoking some functions. The nodes that would have been constructed by the first variant, but are not constructed by the second variant, are roots of needed expressions. We will describe how to acquire such a knowledge in a following section. Below, we give an informal account of it for one of our running examples.

Let $t = \textit{length}\ (t_1 \colon t_2)$, for some subexpressions t_1 and t_2, be a state of a computation. Expression t must be derived to head-constructor form, hence it is needed. If t is rewritten to $u = 1 + \textit{length}(t_2)$, then u is needed as well. Operation "$+$" needs both its arguments, thus $\textit{length}(t_2)$ is needed, too. This explains why, in the second case of the multibranch test of function *length*, the nodes labeled by "$+$" and *length* are not constructed.

We adjust the compiler of Fig. 2 to generate the ahead function of an operation. In the right-hand side of each rewrite rule, we tag any node that we are able to determine is needed. When a right-hand side r is interpolated for printing, if n is a node of r labeled by a non-variable symbol f, then if n is tagged, the compiler generates a call to ahead function f, otherwise, the compiler generates the construction of an instance of class f. The syntactic distinction in the generated code is the absence of operator **new** for tagged nodes. The rest of the compiler is unchanged.

The bodies of method **H** of class f and ahead function f are identical, except for where they access the expressions they manipulate. In method **H**, they are subexpressions of THIS, whereas in function f they are subexpressions of the arguments passed in a call. Therefore, we can replace the body of method **H** of class f with an invocation to ahead function f. When efficiency is a concern, the invocation can be inlined, and this gives exactly method **H** of variant 1. We show this encoding of method **H** for our running example in Fig 5.

```
class length (arg) {
  def H = ↓ length (arg)
}
```

Figure 5: *Implementation of class* length *in which method* **H** *invokes ahead function* length *defined in Fig. 4. Identifier "*length*" is overloaded.*

The encoding of Fig. 5 offers an interesting perspective of our rewriting computations. If a node n labeled by operation f of an expression e is needed, we call ahead function f to derive a head-constructor form from the subexpression rooted by n. If we do not know whether n is needed, we represent n with an instance of class f that contains a "promise" to derive a head-constructor form from the subexpression rooted by n. This promise is delivered when method **H** of this instance is invoked.

5. INFERRING NEED

The second variant of the implementation described in the previous section benefits from knowing, at *compile-time*, needed nodes of a state of a computation. To this aim, we tag some nodes of the right-hand sides of some rewrite rules of a program. When a rule is applied in a needed computation, any expression matched by a tagged node is needed.

The scope of this section is to show that a few inference rules suffice to find many needed nodes of many programs.

Rule 1. Let S be a source program and $l \to r$ a rule of S. Then, tag the root of r.

Rule 2. Let S be a source program, $l \to r$ a rule of S, n tagged node of r and f the label of n. Let π be a maximal (most specific) pattern in a branch node of a definitional tree of f that matches $r|_n$, i.e., for some graph homomorphism σ, $\sigma(\pi) = r|_n$, and o the inductive variable of π. Then, tag $\sigma(o)$.

Rule 3. Let S be a source program, $l \to r$ a rule of S and n a tagged node of r labeled by an arithmetic or relational operator on a numeric type. Then, tag the successors of n.

The above rules may tag nodes that are already known to be constructor-rooted. Tagging one such node n is useless, though harmless, since tagging captures the notion that the expression rooted by n must be derived to a head-constructor form, which it already is.

Rule 3 is an *ad-hoc* version of Rule 2. It is motivated by the fact that in most functional logic languages, numeric types have no algebraic definitions. Hence, arithmetic and relational operators on these types are built-in rather than defined by rewrite rules. Rule 3 regards these operators as if they were defined by a large or infinite set of rules with patterns matching every combination of literal values.

For example, going back to our first introductory examples, in the rule (2) of *abs*, we tag the root node of the right-hand side by Rule 1, the node labeled "$<$" by Rule 2, and the right argument of "$<$" by Rule 3. The node labeled by 0 is constructor-rooted. Tagging it would be useless, since it is a value, hence already constructor-rooted. Similarly, in the second rewrite rule of *length*, we tag the root by Rule 1, the node labeled by *length* by Rule 3, and we do not tag the node labeled by 1.

The correctness of tagging for inferring need is stated as follows. Let S be a *source* program, e an expression of S, and A a needed computation of e. Let $l \to r$ be the rule applied in a step $t \to s$ of A, p the root of the redex, and σ the matching homomorphism, i.e., $\sigma(l) = t|_p$. Tagging a node n of r is *correct* iff $t[\sigma(r)]_p|_n$ is a needed subexpression of s.

We can prove the correctness of tagging for the first two inference rules presented above. The proof is actually trivial for the first rule. The proof of the second rule is a consequence of Lemma 6. The third rule originates from looking at the integers as if they were algebraically defined, and at operations on them accordingly. While there is no doubt that the inference is sensible, a proof of its correctness would require a formalization of this viewpoint.

6. BOXING AND UNBOXING

Built-in types, such as integers, floating point numbers, and characters, require some special handling. In a statically-typed implementation language, the type of an integer expression, e.g., $1 + 2$, must be the same as that of its replacement, 3. In our implementation, the type of each node of an expression is a subclass of *Node*, the base class of all nodes, which for ease of reading we did not show in our abstract implementation. Thus, a *literal expression* such as 3, must consist of a "wrapper", or box, that transforms a built-in value into an expression.

The terms "boxing" and "unboxing" refer to the actions of transforming a built-in value into an expression and extracting a built-in value from an expression, respectively. For example, to evaluate $1 + 2$, both operands of the addition will have to be unboxed, the extracted integer values added together, and the result of the addition boxed. In this way, redex and replacement have the same underlying type.

Boxing and unboxing are relatively expensive operations when compared with the cost of an operation on built-in types. Therefore, it is highly desirable to avoid boxing and unboxing whenever possible. The optimized implementation variant avoids boxing and unboxing in many cases. Ahead functions return head-constructor forms that for built-in types such as numbers and characters are boxed literal values. Furthermore, in some cases, ahead functions are nested. This implies that what a nested function produces the nesting function consumes. Thus, the producer can avoid boxing and consequently the consumer can avoid unboxing.

As an example, consider again ahead function *length*. In the second case of the multibranch test of Fig. 4, the call to function *length* is nested within the call to function "$+$". Ahead function *length* returns a had-constructor expression, hence a boxed built-in integer. Ahead function "$+$" takes this expression, knows that it is constructor-rooted, hence a boxed built-in integer, and unboxes it. Therefore, there is an opportunity to avoid boxing and unboxing the expression produced by *length* and consumed by "$+$". Unfortunately, there is also a potential problem. Ahead function "$+$" may be called from other places in the program, and not every call can guarantee that the second argument will be a literal integer, and consequently could be passed already unboxed. The solution is to specialize some ahead functions, such as those discussed in this example. Specialization of ahead functions is the subject of the next section.

Booleans can be defined algebraically, but are also built-in values in both our abstract and concrete implementation languages. Either representation has pros and cons. We will consider them as algebraically defined, but look at them as built-in whenever it is convenient to pass them unboxed.

7. SPECIALIZATION AND INLINING

Rewriting computations, even needed ones, may construct expressions that do not play any role in a computation. Specializing ahead functions avoids the construction of some of these expressions. We begin by showing the problem and its solution in one of our running examples.

Consider again the rule (2) of *abs*. The second variant of our scheme compiles operation *abs* as follows:

```
def $abs(arg)$ =
  $ifthenelse(>(\textbf{new}\ 0, arg), \textbf{new}\ -(arg), arg)$
```

Figure 6: *Ahead function* abs *compiled from the rule of operation* abs *defined in (2).*

where *ifthenelse*, defined in Fig. 7, identifies the ahead function abstracting the rules of (3), "$>$" identifies the ahead function abstracting the usual "greater then or equal to" operator on numbers, and "$-$" identifies the class abstracting the unary minus operator.
Ahead function *abs* constructs the expression **new** $-(arg)$ no matter what the value of its argument is and passes it to ahead function *ifthenelse*. Ahead function *ifthenelse* makes no use of this expression when *arg* evaluates to a non-negative number. Defining a specialized version of *ifthenelse* for the call within ahead function *abs* avoids constructing this expression when it is not needed. The specialized version of any function f is passed any needed argument—by convention already evaluated—and any variable required to produce the returned expression. Constants need not be passed, since they are known at compile-time and can be encoded in the specialized function. Fig. 8 shows the specialization of *abs* according to the above principles. Specialized versions of a function f are generated by the compiler which append a progressive number, "01", "02", ... to f's identifier.

```
def $ifthenelse(arg1, arg2, arg3)$ =
  $arg1$.H match {
    case $False$ ⇒ $arg2$.H
    case $True$  ⇒ $arg3$.H
  }

def $ifthenelse(arg1, arg2, arg3)$ =
  if $arg1$.H.$tonative$ then $arg2$.H else $arg3$.H
```

Figure 7: *Two functionally equivalent versions of ahead function* ifthenelse *compiled from the rules of (3). The first one is generated by the compiler. The second one, hand coded, executes the native* **if·then·else** *construct. Method* tonative *converts expressions* False *and* True *into the corresponding native Boolean values.*

```
def $abs(arg)$ = {
  $arg$.H
  $ifthenelse\_01(>\_01(arg), arg)$
}

def $ifthenelse\_01(arg1, arg2)$ =
  if $arg1.tonative$ then new $-(arg2)$.H  else $arg2$.H

def $>\_01(arg)$ =
  new $(0 > unbox(arg))$
```

Figure 8: *Ahead functions, some of which are specialized, compiled from the rule of "absolute value" defined in (2). The* **if·then·else** *construct in ahead function* ifthenelse_01 *is the native conditional statement of the implementation language, not the* ifthenelse *function defined in (3). Likewise symbol "$>$" in the body of ahead function* $>$_01 *is the native relational operator.*

The compiler can inline calls to non-recursive ahead functions. This is interesting for any specialization of the *ifthenelse* function, particularly if the implementation uses the native **if·then·else** construct. The result is that an **if·then·else** construct of a functional logic language is compiled into an **if·then·else** construct of the implementation language. We will show a more detailed example of this in a case study.

8. A CASE STUDY

Program *nfib* belongs to the *NoFib* benchmark suite [32], a collection of Haskell programs that have been used for benchmarking *GHC* [15].

```
nfib n = if n <= 1 then 1
  else nfib (n-1) + nfib (n-2) + 1
```
(9)

We discuss the compilation of this program in some detail. Initially we tag the right-hand side. The nodes tagged are *ifthenelse*, "$\leqslant$" and n. Therefore we define specialized versions of the two functions. We also specialize *nfib*. Since its argument is needed, the specialized ahead function expects to receive the argument in head-constructor form, hence a literal integer.

The value returned by "$\leqslant$*_01*" is consumed by *ifthenelse_01*. Therefore, it is convenient to look at its type as built-in and exchange an unboxed native Boolean. Finally, we inline the call to "$\leqslant$*_01*". This leads to the code of Fig. 9, where the dots stand for the portion of code not yet compiled. The yet-to-compile portion of code, involves expressions 1 and *nfib* $(n-1)+$ *nfib* $(n-2)+1$. Both are needed in their respective branches of the **if·then·else** construct according to the implementation of ahead function *ifthenelse* in Fig. 7. We discuss only the second expression, which is the most interesting.

```
def ifthenelse_01(arg1, arg2) =
  if arg1 ⩽ 1 then ···
```

Figure 9: *Portion of the specialized ahead function* ifthenelse *called by* nfib. *Ahead function* ifthenelse_01 *is compiled into the native* **if·then·else** *construct and is passed a native Boolean as its first argument.*

We have already determined that the argument of *nfib* is needed. This leads to tagging every node of this expression. We define specialized versions of each function and inline their calls except for *nfib_01*, since it is recursive. Any time a nested call produces an integer needed by the nesting context, the integer is exchanged unboxed. The resulting code is in Fig. 10, where all the arithmetic and relational operators are native.

```
def ifthenelse_01(arg1, arg2) =
  if arg1 ⩽ 1 then 1 else nfib_01(arg2−1) + nfib_01(arg2−2) + 1
```

Figure 10: *Complete specialized ahead function* ifthenelse *called by* nfib_01.

Finally, we inline the call to *ifthenelse_01* from *nfib_01*. Fig. 11 shows ahead function *nfib_01* and its call from class *nfib*. The arguments of every function call in this code are passed by-value, hence evaluated eagerly, and the code allocates only a single node for the entire computation.

```
def nfib_01(arg) =
  if arg ⩽ 1 then 1 else nfib_01(arg−1) + nfib_01(arg−2) + 1

class nfib(arg) {
  def H = ↓ new nfib_01(arg.H.tonative)
}
```

Figure 11: *Complete code of the implementation of* nfib.

The following table compares various activities of the computation of *nfib*(35) performed with and without using ahead functions. The data were collected by instrumenting the concrete implementation described in the next section. The numbers of rewrite steps, node allocations, etc., include a harness that allocates 2 nodes to construct *nfib*(35), starts the execution and collects the result, hence some data items may be off by a few units with respect to the computation of *nfib*(35).

	ahead	rewriting
rewrite steps	1	119,442,811
node allocations	3	238,885,626
pattern matches	1	328,467,726
ahead calls	29,860,703	
exec. time (sec)	0.03	3.49

The column labeled *ahead* refers to the execution of the program in Fig. 11. The column labeled *rewriting* refers to the execution of a program that evaluates by rewriting only. The number of ahead calls counts only calls to *nfib_01*. It does not included in the count the calls to other ahead function such as "+" and *ifthenelse* because these have been inlined and differ greatly in their contribution to wall-clock time. Should we include these, too, in the count, for each ahead function call in the first column there would be exactly one rewrite step in the second column.

9. CONCRETE IMPLEMENTATION

We encoded the abstract implementation in C++. The overall architecture presented in our simplified Scala language consisting of classes with instances abstracting nodes, variables in these instances abstracting the successors of the node, ahead functions, boxing and unboxing, inlining, etc., is faithfully preserved in the C++ implementation. C++ has no pattern matching. To achieve the same functionality, each node has an integer attribute, a token telling whether the node's label is an operation or a constructor and in this case which constructor. The tokens of the constructors of a type are numbered 0, 1, ... In addition, there are a few additional token numbers for the logic part of the implementation, i.e., whether a node is labeled by the *choice* symbol, a failure, or a variable. These kinds of nodes were not discussed in this paper. Pattern matching is implemented by traversing definitional trees as described in [1, Def. 7], where the token attribute of a inductive node is compared with the token attribute of the matching node.

The reduction of a redex by its replacement, which we have denoted with "$\downarrow$" in our simplified Scala code, is a delicate issue. After a replacement is constructed, a reduction consists of redirecting any reference to the redex from the redex to the replacement, an operation called *pointer redirection* [14, Def. 8]. Finding in some state e of a computation all the references to a node n would be prohibitively expensive. A typical approach is to use an indirect address, say p, to n. Any reference that should point to n, points to p instead. A reduction only redirects p from n to the root of the replacement. This approach is suitable for most imperative programming languages. C++, unique among them, offers a more efficient alternative called *placement new*. This feature allows the programmer to construct a new node in the exact memory location previously occupied by another node. We use *placement new* to construct the root of a replacement in the same memory locations occupied by the root of the redex. Thus, no pointer redirection is required for a reduction. Our notation and use of "$\downarrow$", exactly captures the functionality of *placement new*.

Except for the difference in pattern matching and the use of *placement new* for reductions, which is not available in Scala, our C++ implementation is architecturally equivalent to that presented in our simplified Scala language. A functional logic program is encoded in C++ with the help of a macro language. A data constructor is specified as a label identifier and its arity. A defined operation is specified with a structure that closely matches the structure of a definitional tree. The macro expansion performed by the C++ preprocessor produces C++ source code which is then compiled into executable code. More details about this implementation can be found in [9].

10. FUNCTIONAL LOGIC PROGRAMMING

The ultimate goal of our work is the implementation of functional logic languages. The preceeding discussion left out three essential features of this paradigm: higher-order computations, logic (also called *free*) variables, and non-determinism. In this section we show that our approach is compatible with these features.

Higher-order computations are reduced to first-order computation by a transformation called *firstification* that replaces a partial application of a symbol with a full application [35, 39]. This transformation is part of the process that from a functional logic program produces a graph rewriting system taken by the compiler of Fig. 2 and consequently is effortlessly accomodated by our approach.

Logic variables are equivalent to non-deterministic functions [8, 27]. Every functional logic program containing logic variables can be transformed into a similar functional logic program without logic variables as long as the *choice* operation is allowed in the program. Therefore, the only real extension to our treatment to achieve our goal is handling non-determinism.

In this extention, a program is a *LOIS* graph rewriting system as defined in the introduction. A computation is a sequence of deterministic and/or non-deterministic rewrite steps. A deterministic step is the application of a rule of an inductively sequential operation as described earlier. A non-deterministic step is the application of a rule of operation *choice* defined in (1).

Backtracking is a simple strategy for handling non-deterministic steps. It is adopted by the Prolog language [22] and by many implementations of functional logic languages [13, 17, 29]. With backtracking, a non-deterministic step is executed by arbitrarily applying the first rule of *choice*. If and when the computation produces a result, this result is presented to the user with the option to either terminate the computation or roll it back to the state preceding the last application of the first rule of *choice*, if any, and continue the computation with the application of the second rule.

Backtracking requires recording the steps of a computation in order to unroll the computation to a previous state. This requirement is independent of any approach to rewriting. Both variants presented in the paper can be used in conjunction with backtracking without any change. Backtracking is efficient, but incomplete, i.e., unable to produce all the values of some expressions. Complete strategies for handling non-determinism include context cloning, bubbling [6], and pull-tabbing [5, 12]. All these strategies execute both rewrite steps and some graph transformations different from rewrite steps. The interaction of these transformations with both variants presented in this paper will be the subject of future work.

11. RELATED WORK

The focus of our work is an investigation of the role of "need" in computations in inductively sequential graph rewriting systems. We study this class because for this class there is a very strong notion of "need," which has been tied to optimal strategies, and because a minimal extension of this class is a core language for the implementation of functional logic programming languages [5, Sect. 4.2]. The classic notion of need [20, 21] plays a central role in the definition and implementation of complete and theoretically efficient evaluation strategies for declarative programming languages [4]. For this reason, this notion has been extended to other systems, e.g., *necessary set* [37], where one redex of a set is needed, but which redex is typically not known, or *need modulo a non-deterministic choice* [2], where a redex with distinct replacements is needed, but which replacement is typically not known.

The inductively sequential systems are a proper subclass of the orthogonal systems, hence they have the same notion of *needed redex*. The inductively sequential systems are also constructor-based. Because of the constructor discipline [31], the need of a redex is more stringent: not only a needed redex must be reduced, but also it must be derived to a head-constructor form. This is the crucial observation of this paper and the reason why we can compile inductively sequential systems in a form that skips some needed reduction steps.

Our work is motivated by and targeted to the implementation of functional logic computation by rewriting. Recently, various graph operations different from rewrite steps have been proposed for handling non-determinism [5, 6, 12] in functional logic computations. Variant 2 provides yet another graph operation, different from a rewrite step, intended to improve both theoretical and practical efficiencies of a functional logic computation.

The substantial intersection between functional and functional logic programming relates our work to the implementation of non-strict functional languages as well. However, there are profound differences between the implementation of these two paradigms, as the following program shows:

```
loop = loop

f (0,0) = 0
f (_,1) = 1                                   (10)

main = f (loop,1)
```

The evaluation of expression *main* by a Haskell [33], the leading non-strict functional language, does not produce any result. However, the same evaluation by Curry [16], the leading functional logic language, produces the value 1. The reason is that Haskell translates this program into a core language based on extended lambda calculus [25], whereas Curry looks at it as an inductively sequential systems and evaluates the expression by needed rewriting. This design decision is motivated by the fact that functional logic computations must accommodate both narrowing and non-determinism. We conjecture that our work rediscovers for rewriting some techniques long developed for the extended lambda calculus for compiling non-strict functional languages [25, 26]. There is a wide gap between the two formalisms. The underlying semantics differences should be formalized—a problem in itself—for a meaningful comparison.

Our rewriting computations are lazy (in the sense of call-by-need). However, for some programs, such as our case study, variant 2 of the implementation seems to evaluate eagerly (in the sense of call-by-value). We say "seems" because the overall laziness of a functional logic computation is not affected by the strictness of some ahead functions. Therefore, our work has similarity of intent with strictness analysis [30]. However, there are substantial differences. Strictness analysis is concerned with semantics and considers higher-order computations, whereas we are concerned with operational behavior, we consider only first order computations, and our framework is rewriting.

The specialization of ahead functions discussed in Section 7 is a form of partial evaluation [24]. It contributes, in some cases significantly, to the efficiency of computations. The central idea of our work is the discovery that certain reductions can be executed without a redex. We define functions that compute the reduct without an explicit presence of the redex. These ahead functions can frequently be specialized and this is how ahead computations are related to partial evaluation.

Finally, our approach has the same intent as deforestation, but is complementary to it. Deforestation [38] avoids the construction of constructor-rooted expressions that would be quickly taken apart and disposed. This occurs when a function producing one of these expressions is nested within a function consuming the expression. Ahead computations avoid the construction of operation-rooted expressions that would equally be quickly taken apart and disposed. This occurs when an expression to be constructed is needed and, consequently, must be reduced again. Deforestation and ahead computations can be used independently of each other and jointly in the same program.

12. CONCLUSION

We have described an abstract implementation of rewriting in inductively sequential graph rewriting systems. Any reducible expression in this class defines a step, called needed, that must be executed to obtain the expression's value. A distinctive feature of our implementation is that some of these steps are never executed. Instead, effects similar to the execution of these steps are obtained more efficiently by calling certain functions. At the core of our work is a novel notion of *need* more appropriate for the constructor-based systems, which we have shown equivalent to the classic notion.

We have defined an abstract compiler that takes as input the rules of an operation in the form of a definitional tree and produces as output classes and functions that constitute our implementation. We have precisely stated the correctness of our approach and informally proved these statements. For any expression e, our implementation produces a value v iff v is the constructor normal form of e. We have presented in details a case study from a well-known benchmark suite and compared the performance of our implementation against the performance of an implementation of rewriting that executes only needed steps. This example shows a dramatic reduction in the number of steps, in the number of comparisons for pattern matching, and in the number of allocated nodes. We have informally discussed the application of our work for implementing functional logic languages and shown that with backtracking, a well-known technique for handling non-determinism, our work can be used "out of the box".

To conclude, we would like to answer the question posed in the title: Are *Needed* Redexes Really Needed? If a computer implementation of rewriting executes only rewrite steps, the answer is, unsurprisingly, "yes". However, to execute these steps, the computer implementation calls functions or methods, constructs objects, compares variables, etc. The very same actions may be employed for other purposes in addition to executing rewriting steps. Through these actions, some needed redexes can be reduced by a process different from a rewrite step, and some redexes that would have been created, needed and reduced in a rewriting computation are not created and consequently not reduced.

Thus, some needed redexes appear to be less needed than we thought before. However, somewhat paradoxically, these redexes must be needed to be less needed than we thought before.

13. ACKNOWLEDGMENTS

This material is based upon work partially supported by the National Science Foundation under Grant No. CCF-1317249.

14. 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. On the correctness of pull-tabbing. *TPLP*, 11(4-5):713–730, 2011.

[6] 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.

[7] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, July 2000.

[8] 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.

[9] 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`.

[10] 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.

[11] M. Bezem, J. W. Klop, and R. de Vrijer (eds.). *Term Rewriting Systems*. Cambridge University Press, 2003.

[12] 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.

[13] R. Caballero and J. Sánchez, editors. *TOY: A Multiparadigm Declarative Language (version 2.3.1)*, 2007. Available at `http://toy.sourceforge.net`.

[14] 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`.

[15] *GHC, The Glasgow Haskell Compiler*, 2013. Available at `http://www.haskell.org/ghc/`.

[16] M. Hanus, editor. *Curry: An Integrated Functional Logic Language (Vers. 0.8.2)*, 2006. Available at `http://www-ps.informatik.uni-kiel.de/currywiki/`.

[17] M. Hanus, editor. *PAKCS 1.9.1: The Portland Aachen Kiel Curry System*, 2008. Available at `http://www.informatik.uni-kiel.de/~pakcs`.

[18] 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.

[19] M. Hanus, S. Lucas, and A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8, 1998.

[20] G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems, I. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*, pages 395–414. MIT Press, Cambridge, MA, 1991.

[21] G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems, II. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*, pages 415–443. MIT Press, Cambridge, MA, 1991.

[22] ISO. Information technology - Programming languages - Prolog - Part 1, 1995. General Core. ISO/IEC 13211-1, 1995.

[23] 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.

[24] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.

[25] S.L.P. Jones. *The implementation of functional programming languages*. Prentice-Hall international series in computer science. Prentice/Hill International, 1987.

[26] S.L.P. Jones and D.R. Lester. *Implementing Functional Languages*. Prentice Hall International Series in Computer Science. Prentice Hall PTR, 1992.

[27] 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.

[28] 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.

[29] W. Lux, editor. *The M unster Curry Compiler*, 2012. Available at `http://danae.uni-muenster.de/~lux/curry/`.

[30] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proc. 4th Intl. Symp. on Programming*. Springer LNCS 83, 1980.

[31] M. J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.

[32] W. Partain. The NoFib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag.

[33] S. Peyton Jones and J. Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language*, 1999. Available at `http://www.haskell.org/onlinereport/`.

[34] 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.

[35] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Reprinted from the proceedings of the 25th ACM National Conference*, pages 717–740. ACM, 1972.

[36] The Scala language specification. Available at `http://www.scala-lang.org/node/198`, 2011.

[37] R. C. Sekar and I. V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. *Information and Computation*, 104(1):78–109, 1993.

[38] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

[39] D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.