Needed Computations Shortcutting Needed Steps

Sergio Antoy
Computer Science Dept.
Portland State University
Oregon, U.S.A.
antoys@pdx.edu

Jacob Johannsen
Dept. of Computer Science
Aarhus University
Denmark
cnn@cs.au.dk

Steven Libby
Computer Science Dept.
Portland State University
Oregon, U.S.A.
slibby@pdx.edu

We define a compilation scheme for a constructor-based strongly-sequential graph rewriting system which shortcuts some needed steps. The object code, what the compiler produces on input the rewriting system, is another constructor-based graph rewriting system that is normalizing for the original system when using an innermost strategy. Hence, the object code can be easily implemented by eager functions in a variety of programming languages. We then modify our object code in a way that avoids totally or partially the construction of the contracta of some needed steps of a computation. When computing normal forms in this way, both memory consumption and execution time are reduced compared to ordinary rewriting computations in the original system.

1 Introduction

Rewrite systems are models of computations that specify the actions, but not the control. The object of a computation is a graph referred to as an expression. The actions are encoded by rules that define how to replace (rewrite) one expression with another. The goal of a computation is to reach an expression, called a normal form, that cannot be further rewritten.

The rewrite system does not tell which subexpression of an expression object of a computation should be replaced, and its replacement, to reach the goal. As an example, consider the rewrite system (1). The syntax is Curry [16].

\[\text{loop} = \text{loop}\]
\[\text{snd}(-,y) = y\]

(1)

A computation of \(\text{snd}(\text{loop},0)\) terminates with 0 if the second rule of (1) is applied, but goes on forever without making any progress if only the first rule is applied.

A strategy is a policy or algorithm that in a computation defines both which subexpression should be replaced and its replacement. The intended goal of a strategy is to efficiently produce a normal form, when it exists, of an expression. A practical strategy, called needed, is known for the class of the strongly sequential term rewriting systems [18]. This strategy relies on the fact that in every reducible term \(e\) there exists a redex, called needed as well, that is reduced in any computation of \(e\) to a normal form.

The needed strategy is defined and implemented as follows: given an expression \(e\), while \(e\) is reducible, reduce an arbitrarily chosen, needed redex of \(e\). In the systems considered in this paper, finding a needed redex is easy without look-ahead [3]. This strategy is normalizing: if an expression \(e\) has a normal form, the above loop will terminate with that normal form. This strategy is also optimal in the number of reduced redexes for graph (not term) rewriting.

The above outline shows that implementing a needed strategy is a relatively straightforward task. Surprisingly, however, it is possible to shortcut some of the needed steps in the computation. This paper shows how this shortcutting can be introduced into an implementation of a needed strategy.
Terminology and background information are recalled in Sect. 2. The compilation scheme and its properties are in Sect. 3 and 4. The transformation that allows shortcutting needed redexes, and its properties, are in Sect. 5 and 6. Sect. 7 presents two benchmarks and sketches further opportunities to shortcut needed steps. Related work and conclusion are in Sect. 8 and 9, respectively.

2 Preliminaries

A rewrite system is a pair \((\Sigma \cup \mathcal{X}, \mathcal{R})\) in which \(\Sigma = C \uplus D\) is a signature partitioned into constructors and operations (or functions), \(\mathcal{X}\) is a denumerable set of variables, and \(\mathcal{R}\) is a set of rewrite rules defined 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.

An expression is a single-rooted, directed, acyclic graph defined in the customary way [10, Def. 2]. An expression \(e\) is a constructor form (or value) iff every node of \(e\) is labeled by a constructor symbol. Constructor forms are normal forms, but not vice versa, e.g., \(\text{head}([[]])\) where \(\text{head}\) is the usual operation that returns the first element of a (non-empty) list. In a constructor-based system, such expressions are regarded as failures or exceptions rather than results of computations. Likewise, a head-constructor form is an expression whose root node is labeled by a constructor symbol.

A rule is a pair of expressions with the characteristics discussed below. The rules follow the constructor discipline [23]. 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 \(D\) is inductively sequential, i.e., its rewrite rules are organized in a hierarchical structure called a definitional tree [1].

The inductively sequential systems are the intersection [17] of the strongly sequential systems [18] and the constructor-based systems [23]. The objects of a computation are directed acyclic graphs [10, 11] rather than terms. Sharing some subexpressions of an expression is a requirement of functional logic programming [13, 14, 15, 21]. Incidentally, this sharing ensures that needed redexes are never cloned or duplicated during a computation. The difference between acyclic graphs and ordinary terms is marginal and we will point out consequences of this difference when relevant.

A computation of an expression \(t\) is a finite or infinite sequence

\[ t = t_0 \to t_1 \to \ldots \]

such that \(t_i \to t_{i+1}\) is a rewrite step [10, Def. 23]. For all \(i, t_i\) is a state of the computation of \(t\).

Given a rewrite system \(R\), an expression of \(R\) is an expression over the signature of \(R\). When \(s\) is a signature symbol and \(n\) is a natural number, \(s/n\) denotes that \(n\) is the arity of \(s\). When \(t\) and \(u\) are expressions and \(v\) is a variable, \([u/v]\) is the substitution that maps \(v\) to \(u\) and \(t[u/v]\) is the application of \([u/v]\) to \(t\). The reflexive closure of the of the rewrite relation “\(\to\)” is denoted “\(\to^*\)”.

The following notion [7] for inductively sequential systems is key to our work. We abuse the word “needed” because we will show that our notion extends the classic one [18]. Our notion is a binary relation on nodes (or equivalently on the subexpressions rooted by these nodes since they are in a bijection).

**Definition 1.** Let \(R\) be an inductively sequential system, \(e\) an expression of \(R\) rooted by a node \(p\), and \(n\) a node of \(e\). Node \(n\) is needed for \(e\), and similarly is needed for \(p\), iff in any computation 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 \(R\) is needed iff it is needed for some outermost operation-rooted subexpression of \(e\).
Our “needed” relation is interesting only when both nodes are labeled by operation symbols. If $e$ is an expression whose root node $p$ is labeled by an operation symbol, then $p$ is trivially needed for $p$. This holds whether or not $e$ is a redex and even when $e$ is already a normal form, e.g., $\text{head}([])$. In particular, any expression that is not a value has pairs of nodes in the needed relation. Finally, our definition is concerned with reaching a head-constructor form, not a normal form.

Our notion of need generalizes the classic notion [18]. Also, since our systems follow the constructor discipline [24] we are not interested in expressions that do not have a value.

Lemma 1. Let $R$ be an inductively sequential system and $e$ an expression of $R$ derivable to a value. If $e'$ is an outermost operation-rooted subexpression of $e$, and $n$ is both a node needed for $e'$ and the root of a redex $r$, then $r$ is a needed redex of $e$ in the sense of [18].

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 [18].

Lemma 2. Let $R$ be an inductively sequential system, $e$ an expression of $R$, $e_1$, $e_2$ and $e_3$ subexpressions of $e$ such that $n_i$ is the root of $e_i$ and the label of $n_i$ is an operation, for $i = 1, 2, 3$. If $n_3$ is needed for $n_2$ and $n_2$ is needed for $n_1$, then $n_3$ is needed for $n_1$.

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

3 Compilation

For simplicity and abstraction, we present the object code, $C_R$, of $R$ as a constructor-base graph rewriting system as well. $C_R$ has only two operations called head and norm and denoted $H$ and $N$, respectively. The constructors symbols of $C_R$ are all and only the symbols of $R$.

Operation $H$ is defined piecemeal for each operation of $R$. Each operation of $R$ contributes a number of rules dispatched by pattern matching with top to bottom priority as in common functional languages. The rules of $H$ contributed by an operation with definitional tree $T$ are generated by the procedure compile defined in Fig. 1. The intent of $H$ is to take an expression of $R$ rooted by an operation and derive an expression of $R$ rooted by a constructor by performing only needed steps.

The expression “$\{x\}$” embedded in a string denotes interpolation as in modern programming languages, i.e., the argument $x$ is replaced by a string representation of its value.

The notation $t[u]_p$ [8 2.1.8] stands for an expression equal to $t$ in which the subexpression identified by $p$ is replaced by $u$. In procedure compile the notation is used to “wrap” an application of $H$ around the subexpression of the pattern at $o$, the inductive node. An example is the last rule of (4).

The loop at statement 12 is for collapsing rules, i.e., rules whose right-hand side is a variable. When this variable matches an expression rooted by a constructor of $R$, no further application of $H$ is required. Otherwise, $H$ is applied to the contractum. Symbol “abort” is not considered an element of the signature of $C_R$. If any redex is reduced to “abort”, the computation is aborted since it can be proved that the expression object of the computation has no constructor normal form.
Compiling Needed Computations

Figure 1: Procedure `compile` takes a definitional tree of an operation $f$ of $R$ and produces the set of rules of $H$ that pattern match $f$-rooted expressions.

For example, consider the rules defining the operation that concatenates two lists, denoted by the infix identifier `++`:

\[
\begin{align*}
\texttt{[]}++\texttt{y} &= \texttt{y} \\
\texttt{(x:xs)}++\texttt{y} &= \texttt{x:(xs++y)}
\end{align*}
\] (2)

The definitional tree of operation `++` is pictorially represented below. We use variable identifiers to ease understanding, but the identifiers themselves are irrelevant. The only branch node of this tree is the root. The inductive variable of this branch, boxed in the representation, is $x$. The rule nodes of this tree are the two leaves. The actual rewrite rules associated to the rule nodes are not represented—they are already shown in (2). There are no exempt nodes in this tree since operation `++` is completely defined.

\[\begin{array}{c}
\texttt{x++y} \\
\texttt{[]}++\texttt{y} \quad \texttt{(x:xs)}++\texttt{y}
\end{array}\]

Applying procedure `compile` to this tree produces the following output:

\[
\begin{align*}
H(\texttt{[]}++\texttt{[]}) &= \texttt{[]} & \text{compile line #14} \\
H(\texttt{[]}++\texttt{(y:ys)}) &= \texttt{(y:ys)} & \text{compile line #14} \\
H(\texttt{[]}++\texttt{y}) &= H(\texttt{y}) & \text{compile line #15} \\
H(\texttt{(x:xs)}++\texttt{y}) &= \texttt{x:(xs++y)} & \text{compile line #10} \\
H(\texttt{x++y}) &= H(H(\texttt{x++y})) & \text{compile line #04}
\end{align*}
\] (4)

Operation $N$ of the object code is defined by one rule for each symbol of $R$. In the following metarules,
c/n stands for a constructor of R, f/m stands for an operation of R, and $x_i$ is a fresh variable for every $i$.

\[
\begin{align*}
N(c(x_1, \ldots, x_m)) &= c(N(x_1), \ldots, N(x_m)) \\
N(f(x_1, \ldots, x_n)) &= N(H(f(x_1, \ldots, x_n)))
\end{align*}
\]  
(5)

For example, the rules of $N$ for the list constructors and the operation “++” defined earlier are:

\[
\begin{align*}
N([],) &= [] \\
N(x::xs) &= N(x)::N(xs) \\
N(x++y) &= N(H(x++y))
\end{align*}
\]  
(6)

**Definition 2.** The rewrite system consisting of the $H$ rules generated by procedure compile for the operations of $R$ and the $N$ rules generated according to (5) for all the symbols of $R$ is the object code of $R$ and is denoted $C_R$.

For example, we show the computation of $[1]++[2]$ in both $R$ and $C_R$. We use the desugared notation for list expressions to more easily match the patterns of the rules of “++”.

\[
(1:[])++(2:[]) \rightarrow 1:([]++(2:[])) \rightarrow 1:2:[]
\]  
(7)

and

\[
\begin{align*}
N((1:[])++(2:[])) &
\rightarrow N(H((1:[])++(2:[]))) \\
&\rightarrow N(1:([]):N([]++(2:[]))) \\
&\rightarrow 1:N(H([]++(2:[]))) \\
&\rightarrow 1:N(1::N([])) \\
&\rightarrow 1:2:[]
\end{align*}
\]  
(8)

Computation (8) is longer than (7). If all the occurrences of $N$ and $H$ are “erased” from the states of (8), a concept formalized shortly, and repeated states of the computation are removed, the remaining steps are the same as in (7). The introduction and removal of occurrences of $N$ and $H$ in (8), which lengthen the computation, represent the control, what to rewrite and when to stop. These activities occur in (7) too, but are in the mind of the reader rather than explicitly represented in the computation.

### 4 Compilation Properties

$C_R$, the object code of $R$, correctly implements $R$, i.e., computations performed by $C_R$ produce the results of corresponding computations in $R$ as formalized below. Furthermore, $C_R$ implements a needed strategy, i.e., every reduction performed by $C_R$ is a needed reduction in $R$. In this section, we prove these properties of the object code.

Let $Expr$ be the set of expressions over the signature of $C_R$ output by compile on input a rewrite system $R$. The erasure function $\mathcal{E} : Expr \rightarrow Expr$ is inductively defined by:

\[
\begin{align*}
\mathcal{E}(H(t)) &= \mathcal{E}(t) \\
\mathcal{E}(N(t)) &= \mathcal{E}(t) \\
\mathcal{E}(s(t_1, \ldots, t_n)) &= s(\mathcal{E}(t_1), \ldots, \mathcal{E}(t_n)) \text{ for } s/n \in \Sigma_R
\end{align*}
\]  
(9)

Intuitively, the erasure of an expression $t$ removes all the occurrences of $H$ and $N$ from $t$. The result is an expression over the signature of $R$. 
Lemma 3. Let $R$ be an inductively sequential system and $H$ the head function of $C_R$. For any operation-rooted expression $t$ of $R$, $H(t)$ is a redex.

Proof. Let $f/n$ be the root of $t$, and $T$ the definitional tree of $f$ input to procedure compile. The pattern at the root of $T$ is $f(x_1, \ldots, x_n)$, where each $x_i$ is a variable. Procedure compile outputs a rule of $H$ with left-hand side $H(f(x_1, \ldots, x_n))$. Hence this rule, or a more specific one, reduces $t$. \qed

Comparing graphs modulo a renaming of nodes, as in the next proof, is a standard technique \cite[Def. 15]{10} due to the fact that any node created by a rewrite is fresh.

Lemma 4. Let $R$ be an inductively sequential system and $H$ the head function of $C_R$. Let $t$ be an operation-rooted expression of $R$, and $H(t)$ be reduced by an innermost step resulting from the application of a rule $r$ originating from statement 04 of procedure compile. The argument of the application of $H$ introduced by the step is both operation-rooted and needed for $t$.

Proof. Let $T$ be a definitional tree of the root of $t$. Let $\pi$ be the pattern of the branch node $n$ of $T$ from which rule $r$ originates and let $o$ be the inductive node of $\pi$. Since $r$ writes $t$ and $\pi$ is the left-hand side of $r$ modulo a renaming of variables and nodes, there exists a graph homomorphism $\sigma$ such that $t = \sigma(\pi)$. Our convention on the specificity of the rules defining $H$ establishes that no rule textually preceding $r$ in the definition of $H$ rewrites $t$. Since procedure compile traverses $T$ is post-order, every rule of $H$ originating from a node descendant of $n$ in $T$ textually precedes $r$ in the definition of $H$. Let $q = \sigma(\pi_o)$. For each constructor symbol $c/n$ of $R$ of the sort of $\pi_o$, there is a rule of $H$ with argument $\pi[c(x_1, \ldots, x_n)]_o$, where $x_1, \ldots, x_n$ are fresh variables, and this rule textually precedes $r$ in the definition of $H$. Therefore, the label of $q$ is not a constructor symbol, otherwise this rule would be applied to $t$ instead of $r$. Since the step of $t$ is innermost, $q$ cannot be labeled by $H$ either. Thus, the only remaining possibility is that $q$ is labeled by an operation. We now prove that $q$ is needed for $t$. If $n_1$ and $n_2$ are disjoint nodes (neither is an ancestor of the other) of $T$, then the patterns of $n_1$ and $n_2$ are not unifiable. This is because they have different constructors symbols at the node of the inductive variable of the closest (deepest) common ancestor. Thus, since $t = \sigma(\pi)$, only a rule of $R$ stored in a rule node of $T$ below $n$ can rewrite (a descendant of) $t$ at the root, if any such a rule exists. All these rules have a constructor symbol at the node matched by $o$, whereas $t$ has an operation symbol at $q$, the node matched by $o$. Therefore, $t$ cannot be reduced (hence reduced to a head-constructor form) unless $t|q$ is reduced to a head-constructor form. Thus, $q$ is needed for $t$. \qed

Consider the evaluation of $H(t)$, where $t$ is $([1]++[2])++[3]$. The rule being applied is the last one of \cite{H} which is output by statement 04 of procedure compile. According to the previous lemma an occurrence of $H$ is applied to the first argument of $t$ which is rooted by operation “++” and is needed for $t$.

Lemma 5. Let $R$ be an inductively sequential system and $H$ the head function of $C_R$. Let $t$ be an operation-rooted expression of $R$ and let $A$ denote an innermost finite or infinite computation $H(t) = e_0 \rightarrow e_1 \rightarrow \ldots$ in $C_R$.

1. For every index $i$ in $A$, $\mathcal{E}(e_i) \rightarrow_\mathcal{E}(e_{i+1})$ in $R$.

2. If $A$ terminates (it neither aborts nor is infinite) in an expression $u$, then $u$ is a head-constructor form of $R$.

Proof. Claim 1: Let $r$ be the rule of $H$ applied in the step $e_i \rightarrow e_{i+1}$. There are 3 cases for the origin of $r$. If $r$ originates from statement 04 of compile, then $\mathcal{E}(e_i) = \mathcal{E}(e_{i+1})$ and the claim holds. Otherwise $r$
originates from one of statements 08, 10, 14 or 15. In all these cases, a subexpression of $e_i$ of the form $H(w)$ is replaced by either $H(u)$ (statements 08 and 15) or $u$ (statements 10 and 14), in which $w$ is an instance of the left-hand side of a rule of $R$ and $u$ is the corresponding right-hand side. Thus, in this case too, the claim holds.

Claim 2: By strong induction on the nesting level of the redexes reduced in $A$. In the base case, since $t$ is operation-rooted, every redex is at the root. By Lemma 3, the rule applied in the last reduction of $A$ must originate from either statements 10 or 14 of compile since in all other cases the replacement would be reducible. In both cases, by the definition of compile, the replacement is a head-constructor form of $R$. In the induction case, we observe that any nested application of $H$ is introduced by a rule originating from statement 04 of procedure compile. By Lemma 4, the argument of such an application is operation-rooted. Hence, by the induction hypothesis we assume the claim for every nested (non-root) redex reduced in $A$. All these redexes are derived to expressions of $R$. For root redexes, the reasoning of the base case applies unchanged.

If $A$ denotes a computation $H(t) = e_0 \rightarrow e_1 \rightarrow \ldots$ in $C_R$, then, by Lemma 5, we denote $E(e_0) \rightarrow E(e_1) \rightarrow \ldots$ with $E(A)$ and—with a slight abuse—we regard it as a computation in $R$. Some expression of $E(A)$ may be a repetition of the previous one, rather than the result of a rewrite step. However, it is more practical to silently ignore these duplicates than filtering them out at the expenses of a more complicated definition. We will be careful to avoid an infinite repetition of the same expression. We extend the above viewpoint to computations of $N(t)$ in $C_R$, where $t$ is any expression of $R$.

**Theorem 1.** Let $R$ be an inductively sequential system and $H$ the head function of $C_R$. Let $t$ be an operation-rooted expression of $R$ and let $A$ denote an innermost finite or infinite computation $H(t) = e_0 \rightarrow e_1 \rightarrow \ldots$ in $C_R$. Every step of $E(A)$ is needed.

**Proof.** We prove that for every index $i$ such that $e_i$ is a state of $A$, every argument of an application of $H$ in $e_i$ is needed for $E(e_i)$. Preliminarily, we define a relation “$\prec$” on the nodes of the states of $E(A)$ as follows. Let $p$ and $q$ be nodes of states $E(e_i)$ and $E(e_j)$ of $E(A)$ respectively. We define $p \prec q$ iff $i < j$ or $i = j$ and the expression at $q$ is a proper subexpression of the expression at $p$ in $E(e_i)$. Relation “$\prec$” is a well-founded ordering with minimum element the root of $t$. The proof of the theorem is by induction on “$\prec$”. Base case: Directly from the definition of “need”, since $t$ is rooted by an operation of $R$. Ind. case: Let $q$ be the root of the argument of an application of $H$ in $e_j$ for $j > 0$. We distinguish whether $q$ is the root of the argument of an application of $H$ in $e_{j-1}$. If it is, then the claim is a direct consequence of the induction hypotheses. If it is not, we distinguish whether the step $e_{j-1} \rightarrow e_j$ is an application of a rule $r$ generated either by one of the statements 08 or 15 or by statement 04 of procedure compile. In the first case, $q$ is the root of the replacement of the redex matched by $r$ which by the induction hypothesis is needed for $E(e_{j-1})$. Node $q$ is still labeled by an operation, hence it is needed for $E(e_j)$ directly by the definition of “need”. In the second case, there is a node $p$ of $E(e_j)$ that by the induction hypothesis is needed for $E(e_j)$ and matches the pattern $\pi$ of the branch node of a definitional tree from which rule $r$ originates. Let $q$ be the node of the subexpression of $e_j$ rooted by $p$ matched by $\pi$ at $o$. By Lemma 4, $q$ is needed for $p$. Since $p$ is needed for $E(e_j)$, by Lemma 2 $q$ is needed for $E(e_j)$ and the claim holds in this case, too.

**Corollary 1.** Let $R$ be an inductively sequential system and $C_R$ the object code of $R$. Let $t$ be an expression of $R$ and let $A$ denote an innermost finite or infinite computation $N(t) = e_0 \rightarrow e_1 \rightarrow \ldots$ in $C_R$. Every step of $E(A)$ is needed.
Proof. Operation \( N \) of \( C_R \) applied to an expression \( t \) of \( R \) applies operation \( H \) to every outermost operation-rooted subexpression of \( t \). All these expressions are needed by Def. [1] The claim is therefore a direct consequence of Th. [1]. \( \square \)

Corollary 2. Let \( R \) be an inductively sequential system and \( C_R \) the object code of \( R \). For all expressions \( t \) and constructor forms \( u \) of \( R \), \( t \xrightarrow{*} u \) in \( R \) iff \( N(t) \xrightarrow{*} u \) in \( C_R \) modulo a renaming of nodes.

Proof. Let \( A \) denote some innermost computation of \( N(t) \). Observe that if \( A \) terminates in a constructor form \( u \) of \( R \), then every innermost computation of computation of \( N(t) \) terminates in \( u \). Therefore, we consider whether \( A \) terminates normally. Case 1: \( A \) terminates normally. If \( N(t) \xrightarrow{*} u \), then by Lemma [5] point 1, \( t \xrightarrow{*} u \). Case 2: \( A \) does not terminate normally. We consider whether \( A \) aborts. Case 2a: \( A \) aborts. Suppose \( N(t) = e_0 \rightarrow e_1 \rightarrow \ldots e_i \), and the step of \( e_i \) reduces a redex \( r \) to “abort”. By Th. [1] \( r \) is needed for \( e_i \), but there is no rule in \( R \) that reduces \( r \), hence \( t \) has no constructor form. Case 2b: \( A \) does not terminates. Every step of \( \mathcal{E}(A) \) is needed. The complete tree unraveling [8, Def. 13.2.9] of the rules of \( R \) and the states of \( \mathcal{E}(A) \), gives an orthogonal term rewriting system and a computation of the unraveled \( t \). Since redex are innermost, in this computation an infinite number of needed redexes are reduced. This computation, “fair for needed redexes” [18, p. 413], is normalizing, hence \( t \) has no constructor form. \( \square \)

The object code \( C_R \) for a rewrite system \( R \) is subjectively simple. Since innermost reductions suffice for the execution, operations \( H \) and \( N \) can be coded as functions that take their argument by-value, a parameter passing mode which is available and efficient in most programming languages. Corollary[2] in conjunction with Theorem[1] shows that \( C_R \) is a good object code: it produces the value of an expression \( t \) when \( t \) has such value, and it produces this value making only steps that must be made by any rewrite computation. One could infer that there cannot be a substantially better object code, but this is not true. The next section discusses why.

5 Transformation

We transform the object code to avoid totally or partially the construction of certain contracta. The transformation consists of two phases.

The first phase replaces certain rules of \( H \). Let \( r \) be a rule of \( H \) in which \( H \) is recursively applied to a variable, say \( x \), as in the third rule of (4). Rule \( r \) is replaced by the set \( S_r \) of rules obtained as follows. A rule \( r_f \) is in \( S_r \) iff \( r_f \) is obtained from \( r \) by instantiating \( x \) with \( f(x_1, \ldots, x_n) \), where \( f/n \) is an operation of \( R \), \( x_1, \ldots, x_n \) are fresh variables, and the sorts of \( f(x_1, \ldots, x_n) \) and \( x \) are the same. If a rule in \( S_r \) still applies \( H \) to another variable, it is again replaced in the same way.

For example, the following rule originates from instantiating \( y \) for operation ++ in the third rule of (4).

\[
H([]++(u++v)) = H(u++v)
\]

The first phase of the transformation ensures that \( H \) is always applied to an expression rooted by some operation \( f \) of \( R \). The second phase introduces, for each operation \( f \) of \( R \), a new operation, denoted \( H_f \), which is the composition of \( H \) with \( f \), and then replaces every occurrence of the composition of \( H \) with \( f \) with \( H_f \).

For example, (10) becomes:

\[
H_{++}([], u++v) = H_{++}(u, v)
\]
After the second phase, operation $H$ can be eliminated from the object code since it is no longer invoked. We denote the transformed $C_R$ with $T_R$.

$T_R$ is more efficient than $C_R$ because, for any operation $f$ of $R$, the application of $H_f$ avoids the allocation of a node labeled by $f$ which instead is allocated by $C_R$. This node is also likely to be pattern matched later. Below we show the traces of a portion of the computations of $N(([]++[])+t)$ executed by $C_R$ (left) and $T_R$ (right), where $t$ is an irrelevant expression.

<table>
<thead>
<tr>
<th>$N(H([]++[])+t))$</th>
<th>$N(Ht)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N(HH([],[])+t))$</td>
<td>$N(Ht)$</td>
</tr>
<tr>
<td>$N(H[]+t))$</td>
<td>$N(H([],[]),t))$</td>
</tr>
</tbody>
</table>

$C_R$ constructs the expression rooted by the underlined occurrence of “++”, and later pattern matches it. The same expression is neither constructed nor pattern matched by $T_R$.

### 6 Transformation Properties

We show that both phases of the transformation described in the previous section preserve the object code computations.

**Lemma 6.** Let $R$ be an inductively sequential system and $C_R$ the object code of $R$. Let $C'_R$ the rewrite system obtained from $C_R$ according to phase 1 of the transformation. Every step of $C_R$ is a step of $C'_R$ and vice versa, modulo a renaming of nodes.

**Proof.** Let $t \rightarrow u$ be a step of $C_R$ where some rule $r$ is applied. It suffices to consider the case in which $t$ is the redex and the rule $r$ applied in the step is not in $C'_R$. In this case, the right-hand side of $r$ introduces an application of $H$ “around” a variable, say $v$, of $t$. Rule $r$ is output by statement either 04 or 15 of procedure compile. In both cases, the subexpression of $t$ matched by $v$, say $s$, is operation-rooted. If $r$ is output by statement 04, this property is ensured by Lemma 4. If $r$ is output by statement 15, and the match of $v$ were constructor-rooted, then some rule output by statement 14 of procedure compile, which textually precedes $r$ and is tried first, would match $t$. Therefore, let $f/n$ be the root of $s$. By the definition of phase 1 of the transformation, rule $r[f(x_1,\ldots,x_n)/v]$ is in $C'_R$. Therefore, modulo a renaming of nodes, $t \rightarrow u$ in $C'_R$. The converse is similar.

### Corollary 3.** Let $R$ be an inductively sequential system and $C_R$ the object code of $R$. Let $C'_R$ the rewrite system obtained from $C_R$ according to phase 1 of the transformation and $T_R$ the rewrite system obtained from $C_R$ according to phase 2 of the transformation. For every operation-rooted expression $t$ and head-constructor form $u$ of $R$, $H(t) \rightarrow u$ in $C'_R$ iff $\tau(H(t)) \rightarrow u$ in $T_R$ modulo a renaming of nodes.

**Proof.** Throughout this proof, expression equality is intended modulo a renaming of nodes. Preliminarily, we show that for all expressions $t$ and $u$, $t \rightarrow u$ in $C'_R$ iff $\tau(t) \rightarrow \tau(u)$ in $T_R$. Assume $t \rightarrow u$ in $C'_R$. There exists a rule $l \rightarrow r$ of $C'_R$ and a match $\sigma$ such that $t = \sigma(l)$ and $u = \sigma(r)$. From the definition of phase 2 of the transformation, $\tau(l) \rightarrow \tau(r)$ is a rule of $T_R$. Hence, there is an expression $v$ and a match $\sigma'$ such that $\tau(t) = \sigma'(\tau(l)) \rightarrow \sigma'('\tau(r)) = v$ in $T_R$. Since $\tau$ is the identity on variables, $H$ is never applied to a variable, and $\sigma$ is the identity on non variables, $\sigma \circ \tau = \tau \circ \sigma$ and $\sigma = \sigma'$. Therefore $\tau(t) = \sigma(\tau(l))$ and $v = \sigma(\tau(r))$. Hence, $v = \sigma(\tau(r)) = \tau(\sigma(r)) = \tau(u)$. The converse is similar.

Now, we prove the main claim. An induction on the length of $H(t) \rightarrow u$ in $C'_R$ shows that $\tau(H(t)) \rightarrow \tau(u)$ in $T_R$. Since by assumption $u$ is an expression of $R$, by the definition of $\tau$, $\tau(u) = u$. 


Finally, we prove that object code and transformed object code execute the same computations.

**Theorem 2.** Let $R$ be an inductively sequential system, $C_R$ the object code of $R$ and $T_R$ the transformed object code. For all expressions $t$ and $u$ of $R$, $N(t) \rightarrow u$ in $C_R$ iff $N(t) \rightarrow u$ in $T_R$.

**Proof.** In the computation of $N(t)$ in $C_R$, by the definition of $\tau$, each computation of $H(v)$ in $C_R$, for some expression $v$, is transformed into a computation of $\tau(H(v))$ in $T_R$. By Lemma 5, the former ends in a head-construct form of $R$. Hence, by Cor. 3, $\tau(H(v))$ ends in the same head-construct form of $R$. Thus, $N(t) \rightarrow u$ in $T_R$ produces the same result. The converse is similar. \qed

## 7 Benchmarking

Our benchmarks use integer values. To accommodate a built-in integer in a graph node, we define a kind of node whose label is a built-in integer rather than a symbol. An arithmetic operation, such as the addition, retrieves the integers from these nodes, adds them together, and places back the result in a node.

Our first benchmark evaluates $\text{length}(l_1++l_2)$, where $\text{length}$ is the usual length–of–a–list operation and $l_1$ and $l_2$ are lists of equal length. We present a portion of the object code originating from operation $\text{length}$:

\[
\begin{align*}
H(length([])) &= 0 \\
H(length(\_:xs)) &= H(1+length(xs)) \\
&\ldots
\end{align*}
\]

The corresponding transformed rules are:

\[
\begin{align*}
H_{\text{length}}([],[]) &= 0 \\
H_{\text{length}}(\_:xs) &= H_{\#}(1, length(xs)) \\
&\ldots
\end{align*}
\]

In the table below, we compare the same rewriting computation executed by $C_R$ and $T_R$. We measure the number of rewrite and shortcut steps executed, the number of nodes allocated, and the number of node labels compared by pattern matching. We do not report execution times, though $T_R$ is always significantly faster than $C_R$. The tabular entries are normalized with respect to the number of rewrite steps of $C_R$ and are constant functions of this value except for very short lists. For lists of one million elements, the number of rewrite steps of $C_R$ is two millions.

<table>
<thead>
<tr>
<th>$length(l_1++l_2)$</th>
<th>$C_R$</th>
<th>$T_R$</th>
<th>$O_R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>rewrite steps</td>
<td>10</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>shortcut steps</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>node allocations</td>
<td>20</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>node matches</td>
<td>40</td>
<td>26</td>
<td>18</td>
</tr>
</tbody>
</table>

The column labeled $O_R$ refers to object code that further shortcuts needed steps using the same idea behind the transformation. For example, in the second rule of (12) both arguments of the addition in the right-hand side are needed. This information is known at compile-time, hence the compiler can wrap an application of $H$ around the right operand of “+” in the right-hand side of the rule.

\[
H(length(\_:xs)) = H(1+H(length(xs)))
\]

\[ (14) \]
The composition of $H$ with $\text{length}$ is replaced by $H_{\text{length}}$ during the second phase. The resulting rule is:

$$H_{\text{length}}(\text{\_}::\text{xs}) = H_*(1, H_{\text{length}}(\text{xs}))$$  \hspace{1cm} (15)

Of course, there is no need to allocate a node for expression 1, the first argument of the addition, every time rule (13) or (15) is applied, since a single node can be shared by the entire computation. In this case the application of rule (15) would allocate no node of the contractum in practice skipping the step. In our benchmarks, we ignore any optimization that is not directly related to the method we are presenting. Thus $C_R$, $T_R$ and $O_R$ needlessly allocate this node every time these rules are applied.

The number of shortcut steps of $T_R$ and $O_R$ remain the same because, loosely speaking $O_R$ shortcuts a step that was already shortcut by $T_R$, but the number of nodes allocated and matched further decrease. The effectiveness of $T_R$ to reduce nodes allocation or pattern matching with respect to $C_R$ vary with the program and the computation. Our second benchmark concerns a program that computes Fibonacci numbers:

$$\begin{align*}
\text{fib} & \ 0 = 0 \\
\text{fib} & \ 1 = 1 \\
\text{fib} & \ n = \text{fib} (n-1) + \text{fib} (n-2)
\end{align*}$$  \hspace{1cm} (16)

Since pattern matching is performed scanning the rules in textual order, the last rule is applied only when the argument of fib is neither 0 nor 1.

<table>
<thead>
<tr>
<th>$\text{fib}(n)$</th>
<th>$C_R$</th>
<th>$T_R$</th>
<th>$O_R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>rewrite steps</td>
<td>10</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>shortcut steps</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>node allocations</td>
<td>24</td>
<td>22</td>
<td>10</td>
</tr>
<tr>
<td>node matches</td>
<td>44</td>
<td>26</td>
<td>16</td>
</tr>
</tbody>
</table>

The tabular entries are again normalized with respect to the number or steps of $C_R$ and are constant functions of this value except for very small arguments of fib. For $n = 32$, the number of steps of $C_R$ is 17,622,886. With respect to $C_R$, $T_R$ avoids the construction of the root of the right-hand side of the third rule of (16). $O_R$ transforms the right-hand side of this rule into:

$$H_*(H_{\text{fib}}(H-(\text{n},1)), H_{\text{fib}}(H-(\text{n},2)))$$  \hspace{1cm} (17)

since every node that is not labeled by the variable or the constants 1 and 2 is needed. Of course, in this benchmark too, there is no need to allocate a node for either 1 or 2 every time (17) is constructed/executed, but we are concerned only with optimizations related to our approach. With this further optimization, the step would be skipped and the relative gains of our approach would be even more striking.

8 Related Work

The redexes that we reduce are needed to obtain a constructor-rooted expression, hence they are closely related to the notion of root-need of \cite{22}. However, we are interested only in normal forms that are constructor forms. In contrast to a computation according to \cite{22}, our object code may abort the computation of an expression $e$ if no constructor normal form of $e$ is reachable, even if $e$ has a needed redex. This is a
very desirable property in our intended domain of application since it saves useless rewrite steps and in some cases may lead to the termination of an infinite computation.

Machines for graph reduction have been proposed [9,20] for the implementation of functional languages. While there is a commonality of intent, these efforts differ from ours in two fundamental aspects. Our object code is easily translated into a low-level language like C or assembly, whereas these machines have instructions that resemble those of an interpreter. There is no explicitly notion of need in the computations of performed by these machines. Optimizations of these machines are directed toward their internal instructions, rather than the needed steps of a computation by rewriting, a problem less dependent on any particular mechanism used to compute a normal form.

A compilation scheme similar to ours is described in [2]. This effort makes no claims of correctness, of executing only needed steps and of shortcutting needed steps. Transformations of rewrite systems for compilation purposes are described in [12,19]. These efforts are more operational than ours. A compilation with the same intent as ours is described in [7]. The compilation scheme is different. This effort does not claim to execute only needed steps, though it shortcuts some of them. Shortcutting is obtained by defining ad-hoc functions whereas we present a formal systematic way through specializations of the head function.

9 Conclusion

Our work is motivated by the implementation of functional logic computations, which can be formalized and modeled by graph rewriting. The graph rewriting systems modeling functional logic programs are not orthogonal for the presence of overlapping rules and extra variables which occur in the right-hand sides of rules, but not the left-hand sides. Our approach can be applied in this situation in conjunction with techniques that eliminate the extra variables [6] and avoid the application of overlapping rules [4,5] without affecting the results of computations. Future work will systematically investigate opportunities to shortcut needed steps in situations similar to that discussed for operation length in Sect. 7.

Acknowledgments

This material is based upon work partially supported by the National Science Foundation under Grant No. CCF-1317249. This work was carried out while the second author was visiting University of Oregon, Oregon, USA. The second author wishes to thank Zena Ariola for hosting this visit. The authors wish to thank Olivier Danvy for insightful comments.

References


