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 is another constructor-based graph rewriting system. This system is normalizing for the original system when using an innermost strategy. Consequently, the object code can be easily implemented by eager functions in a variety of programming languages. We modify this object code in a way that avoids total or partial 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.

In the computation of an expression, a rewrite system does not tell which subexpression should be replaced to reach the goal.

Example 1. *Consider the following rewrite system. The syntax is Curry [17].*

```
loop = loop
snd (_,y) = y                                                    (1)
```

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

In a computation a *strategy* is a policy or algorithm that defines both which subexpression should be replaced and its replacement. The intended goal of a strategy is to efficiently produce a normal form of an expression when it exists. 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 expression e, there exists a *redex*, also called *needed*, 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, repeatedly reducing arbitrary needed redexes 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.

A. Middeldorp and F. van Raamsdonk (Eds.): 8th International Workshop on Computing with Terms and Graphs (TERMGRAPH 2014)
EPTCS 183, 2015, pp. 18–32, doi:10.4204/EPTCS.183.2

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. Sect. 8 discusses the application of our work to the implementation of functional logic languages. Related work and conclusion are in Sect. 9 and 10, respectively.

2 Preliminaries

A *rewrite system* is a pair $(\Sigma \cup \mathscr{X}, \mathscr{R})$ in which $\Sigma = \mathscr{C} \uplus \mathscr{D}$ is a *signature* partitioned into *constructors* and *operations* (or functions), $\mathscr{X}$ is a denumerable set of *variables*, and $\mathscr{R}$ is a set of *rewrite rules* defined below. Without further mention, we assume that the symbols of the signature have a type, and that any expression over the signature is well typed.

An *expression* is a single-rooted, directed, acyclic *graph* defined in the customary way [11, Def. 2]. An expression e is a *constructor form* (or *value*) if, and only if, every node of e is labeled by a constructor symbol. Constructor forms are normal forms, but not vice versa. For example, $head([\,])$ where $head$ is the usual operation that returns the first element of a (non-empty) list, is a normal form, but not a constructor form. 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 graph with two roots abstracting the left- and right-hand sides respectively. 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 only *constructor* symbols and variables. Rules are left-linear, i.e., the left-hand side is a tree.

The objects of a computation are graphs 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 duplicated during a computation. The difference between our graphs and ordinary terms concerns only the sharing of subexpressions.

A *computation* of an expression t is a possibly infinite sequence

$$t = t_0 \to t_1 \to \ldots$$

such that $t_i \to t_{i+1}$ is a rewrite step [11, 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 rewrite relation "$\to$" is denoted "$\stackrel{=}{\to}$".

Each operation in $\mathscr{D}$ is *inductively sequential*; that is, its rewrite rules are organized into a hierarchical structure called a *definitional tree* [1] which we informally recall below. An example of a definitional tree is shown in (3). In a definitional tree of an operation f, there are up to 3 kinds of nodes called *branch*, *rule* and *exempt*. Each kind contains a pattern of f and other items of information depending on the kind. A *rule* node with pattern π contains a rule of f whose left-hand side is equal to π modulo renaming nodes and variables. An *exempt* node with pattern π contains no other information. There is no rule of f whose left-hand side is equal to π. A *branch* node with a pattern π contains children that are subtrees of the definitional tree. At least one child is a *rule* node. The children are obtained by "narrowing" pattern π. Let x be any variable of π, which is called *inductive*. For each constructor c/m of the type of x, there is a child whose pattern is obtained from π by instantiating x with $c(x_1, \ldots x_m)$, where x_i is a fresh variable.

An operation f/n is *inductively sequential* [1] if there exists a definitional tree whose root has pattern $f(x_1, \ldots x_n)$, where x_i is a fresh variable, and whose leaves contain all, and only, the rules of f. A rewrite system is *inductively sequential* if all of its operations are inductively sequential.

Inductively sequential operations can be thought of as "well designed" with respect to evaluation. To compute a needed redex of an expression e rooted by an operation f, match to e the pattern π of a maximal (deepest in the tree) node N of a definitional tree of f. If N is an *exempt* node, e has no constructor normal form, and the computation can be aborted. If N is a *rule* node, e is a redex and can be reduced by the rule in N. If N is a *branch* node, let x be the inductive variable of π and t the subexpression of e to which x is matched. Then, recursively compute a needed redex of t.

The inductively sequential systems are the intersection [16] of the strongly sequential systems [18] and the constructor-based systems [23]. The following notion [8] 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, if, and only if, 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 *if, and only if, 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., $head([\,])$. 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 [23] 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, any node in any path from the root of e to the root of e', except for the root of e', is 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. □

```
compile $\mathcal{T}$
01  case $\mathcal{T}$ of
02  when $branch(\pi, o, \bar{\mathcal{T}})$ then
03     $\forall \mathcal{T}_i \in \bar{\mathcal{T}}$ compile $\mathcal{T}_i$
04     output "$\mathbf{H}(\$\{\pi\}) = \mathbf{H}(\$\{\pi[\mathbf{H}(\pi|_o)]_o\})$"
05  when $rule(\pi, l \to r)$ then
06     case $r$ of
07        when operation-rooted then
08           output "$\mathbf{H}(\$\{l\}) = \mathbf{H}(\$\{r\})$"
09        when constructor-rooted then
10           output "$\mathbf{H}(\$\{l\}) = \$\{r\}$"
11        when variable then
12           for each constructor $c/n$ of the sort of $r$
13              let $l' \to r' = (l \to r)[c(x_1, \ldots x_n)/r]$
14              output "$\mathbf{H}(\$\{l'\}) = \$\{r'\}$"
15           output "$\mathbf{H}(\$\{l\}) = \mathbf{H}(\$\{r\})$"
16  when $exempt(\pi)$ then
17     output "$\mathbf{H}(\$\{\pi\}) = \text{abort}$"
```

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

3 Compilation

For simplicity and abstraction, we present the object code, C_R, of R as a constructor-based graph rewriting system as well. C_R has only two operations called *head* and *norm*, and denoted $\mathbf{H}$ and $\mathbf{N}$, respectively. The constructor symbols of C_R are all, and only, the symbols of R. The rules of C_R have a priority established by the textual order. A rule reduces an expression t only if no other preceding rule could be applied to reduce t. These semantics are subtle, since t could become reducible by the preceding rule only after some internal reduction. However, all our claims about computations in C_R are stated for an innermost strategy. In this case, when a rule is applied, no internal reduction is possible, and the semantics of the priority are straightforward.

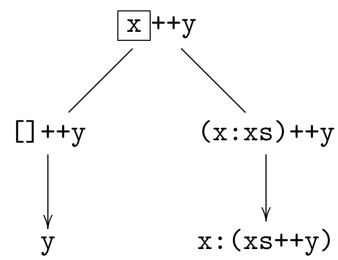
Operation $\mathbf{H}$ is defined piecemeal for each operation of R. Each operation of R contributes a number of rules dispatched by pattern matching with textual priority. The rules of $\mathbf{H}$ contributed by an operation with definitional tree $\mathcal{T}$ are generated by the procedure `compile` defined in Fig. 1. The intent of $\mathbf{H}$ 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 $t[u]_p$ 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 $\mathbf{H}$ 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 $\mathbf{H}$ is required; Otherwise, $\mathbf{H}$ 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.

Example 2. *Consider the rules defining the operation that concatenates two lists, denoted by the infix identifier "++":*

$$\begin{array}{l}\texttt{[]++y = y}\\ \texttt{(x:xs)++y = x:(xs++y)}\end{array} \tag{2}$$

The definitional tree of operation "++" is pictorially represented below. 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. There are no* exempt *nodes in this tree since operation "++" is completely defined.*



(3)

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

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

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

$$\begin{array}{l}\mathbf{N}(c(x_1,\ldots x_m)) = c(\mathbf{N}(x_1),\ldots \mathbf{N}(x_m))\\ \mathbf{N}(f(x_1,\ldots x_n)) = \mathbf{N}(\mathbf{H}(f(x_1,\ldots x_n)))\end{array} \tag{5}$$

Example 3. *The rules of* **N** *for the list constructors and the operation "++" defined earlier are:*

$$\begin{array}{l}\mathbf{N}\texttt{([]) = []}\\ \mathbf{N}\texttt{(x:xs) = }\mathbf{N}\texttt{(x):}\mathbf{N}\texttt{(xs)}\\ \mathbf{N}\texttt{(x++y) = }\mathbf{N}\texttt{(}\mathbf{H}\texttt{(x++y))}\end{array} \tag{6}$$

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.*

Example 4. *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 "++".*

$$\texttt{(1:[])++(2:[])} \to \texttt{1:([]++(2:[]))} \to \texttt{1:2:[]} \tag{7}$$

and

$$
\begin{array}{l}
\mathbf{N}\texttt{((1:[])++(2:[]))} \\
\quad \to \mathbf{N}\texttt{(}\mathbf{H}\texttt{((1:[])++(2:[])))} \\
\quad \to \mathbf{N}\texttt{(1:([]++(2:[]))} \\
\quad \to \mathbf{N}\texttt{(1):}\mathbf{N}\texttt{([]++(2:[]))} \\
\quad \to \texttt{1:}\mathbf{N}\texttt{(}\mathbf{H}\texttt{([]++(2:[])))} \\
\quad \to \texttt{1:}\mathbf{N}\texttt{(2:[])} \\
\quad \to \texttt{1:}\mathbf{N}\texttt{(2):}\mathbf{N}\texttt{([])} \\
\quad \to \texttt{1:2:[]}
\end{array}
\qquad (8)
$$

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. Computations performed by C_R produce the results of corresponding computations in R as formalized below. Furthermore, C_R implements a needed strategy, because 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 $\mathscr{E} : Expr \to Expr$ is inductively defined by:

$$
\begin{array}{l}
\mathscr{E}(H(t)) = \mathscr{E}(t) \\
\mathscr{E}(N(t)) = \mathscr{E}(t) \\
\mathscr{E}(s(t_1, \ldots t_n)) = s(\mathscr{E}(t_1), \ldots \mathscr{E}(t_n)) \;\text{ for }\; s/n \in \Sigma_R
\end{array}
\qquad (9)
$$

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,* $\mathbf{H}(t)$ *is a redex.*

Proof. Let f/n be the root of t, and $\mathscr{T}$ the definitional tree of f input to procedure `compile`. The pattern at the root of $\mathscr{T}$ 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 $\mathbf{H}(f(x_1, \ldots x_n))$. Hence this rule, or a more specific one, reduces t. □

Comparing graphs modulo a renaming of nodes, as in the next proof, is a standard technique [11] 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* $\mathbf{H}(t)$ *be reduced by a step resulting from the application of a rule r originating from statement 04 of procedure* `compile`*. The argument of the inner application of* **H** *in the contractum is both operation-rooted and needed for t.*

Proof. Let $\mathscr{T}$ be a definitional tree of the root of t. Let π be the pattern of the *branch* node n of $\mathscr{T}$ from which rule r originates and let o be the inductive node of π. Since r rewrites t and π is the left-hand side of r modulo a renaming of variables and nodes, there exists a graph homomorphism σ 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 $\mathbf{H}$ rewrites t. Since procedure `compile` traverses $\mathscr{T}$ in post-order, every rule of $\mathbf{H}$ originating from a node descendant of n in $\mathscr{T}$ textually precedes r in the definition of $\mathbf{H}$. Let $q = \sigma(\pi|_o)$. For each constructor symbol c/n of R of the sort of $\pi|_o$, there is a rule of $\mathbf{H}$ with argument $\pi[c(x_1,\dots,x_n)]|_o$, where $x_1,\dots,x_n$ are fresh variables, and this rule textually precedes r in the definition of $\mathbf{H}$. 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 $\mathbf{H}$ 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 $\mathscr{T}$, 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 $\mathscr{T}$ 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. □

Example 5. *The situation depicted by the previous lemma can be seen in the evaluation of* $t =$ `([1]++[2])++[3]`. *According to (4),* $\mathbf{H}(t) \to \mathbf{H}($`H([1]++[2])++[3]`$)$. *The argument of the inner application of* $\mathbf{H}$ *is both operation-rooted and needed for* t.

Lemma 5. *Let R be an inductively sequential system and $\mathbf{H}$ 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 $\mathbf{H}(t) = e_0 \to e_1 \to \dots$ in C_R.*

1. *For every index i in A, $\mathscr{E}(e_i) \overset{=}{\to} \mathscr{E}(e_{i+1})$ 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 $l \to r$ be the rule of $\mathbf{H}$ applied in the step $e_i \to e_{i+1}$. There are 3 cases for the origin of $l \to r$. If $l \to r$ originates from statement 04 of `compile`, then $\mathscr{E}(e_i) = \mathscr{E}(e_{i+1})$ and the claim holds. Otherwise $l \to r$ originates from one of statements 08, 10, 14 or 15. In all these cases, a subexpression of e_i of the form $\mathbf{H}(w)$ is replaced by either $\mathbf{H}(u)$ (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: If A aborts or does not terminate, the claim is vacuously true. So, consider the last step of A. This step cannot originate from the application of a rule that places $\mathbf{H}$ at the root of the contractum, since another step would become available. Hence the rule of the last step is generated by statement 10 or 14 of procedure `compile`. In both cases, the contractum is a head constructor form. □

If A denotes a computation $\mathbf{H}(t) = e_0 \to e_1 \to \dots$ in C_R, then, by Lemma 5, we denote $\mathscr{E}(e_0) \overset{=}{\to} \mathscr{E}(e_1) \overset{=}{\to} \dots$ with $\mathscr{E}(A)$ and—with a slight abuse—we regard it as a computation in R. Some expression of $\mathscr{E}(A)$ 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 $\mathbf{N}(t)$ in C_R, where t is any expression of R.

Theorem 1. *Let R be an inductively sequential system and $\mathbf{H}$ 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 $\mathbf{H}(t) = e_0 \to e_1 \to \dots$ in C_R. Every step of $\mathscr{E}(A)$ 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 $\mathbf{H}$ in e_i is needed for $\mathscr{E}(e_i)$. Preliminarily, we define a relation "$\prec$" on the nodes of the states of $\mathscr{E}(A)$ as follows. Let p and q be nodes of states $\mathscr{E}(e_i)$ and $\mathscr{E}(e_j)$ of $\mathscr{E}(A)$ 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 $\mathscr{E}(e_i)$. 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. Induction case: Let q be the root of the argument of an application of $\mathbf{H}$ in e_j for $j > 0$. We distinguish whether q is the root of the argument of an application of $\mathbf{H}$ in e_{j-1}. If it is, then the claim is a direct consequence of the induction hypothesis. If it is not, $e_{j-1} \to e_j$ is an application of a rule r generated by one of the statements 04, 08 or 15 of procedure `compile`. For statement 04, there is a node p of $\mathscr{E}(e_j)$ that by the induction hypothesis is needed for $\mathscr{E}(e_j)$ and matches the pattern π 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 π at o. By Lemma 4, q is needed for p. Since p is needed for $\mathscr{E}(e_j)$, by Lemma 2, q is needed for $\mathscr{E}(e_j)$ and the claim holds. For statements 08 and 15, q is the root of the contractum of the redex matched by r which by the induction hypothesis is needed for $\mathscr{E}(e_{j-1})$. Node q is still labeled by an operation, hence it is needed for $\mathscr{E}(e_j)$ directly by the definition of "need". □

Corollary 1. *Let R be an inductively sequential system. Let t be an expression of R and let A denote an innermost finite or infinite computation $\mathbf{N}(t) = e_0 \to e_1 \to \ldots$ in C_R. Every step of $\mathscr{E}(A)$ is needed.*

Proof. Operation $\mathbf{N}$ of C_R applied to an expression t of R applies operation $\mathbf{H}$ 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. □

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

Proof. Let A denote some innermost computation of $\mathbf{N}(t)$. Observe that if A terminates in a constructor form u of R, then every innermost computation of $\mathbf{N}(t)$ terminates in u because the order of the reductions is irrelevant. Therefore, we consider whether A terminates normally. Case 1: A terminates normally. If $\mathbf{N}(t) \xrightarrow{*} u$, 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 $\mathbf{N}(t) = e_0 \to e_1 \to \ldots \to e_i$, and the step of e_i reduces a redex r to "abort". By Theorem 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 $\mathscr{E}(A)$ is needed. The complete tree unraveling [9, Def. 13.2.9] of the rules of R and the states of $\mathscr{E}(A)$, gives an orthogonal *term* rewriting system and a computation of the unraveled t. Since redexes are innermost, in this computation an infinite number of needed redexes are reduced. The hypernormalization of the needed strategy [9, Sect. 9.2.2] shows that hence t has no constructor form. □

The object code C_R for a rewrite system R is subjectively simple. Since innermost reductions suffice for the execution, operations $\mathbf{H}$ and $\mathbf{N}$ can be coded as functions that take their argument by-value. This is 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, constructing 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 $f(x_1,\dots x_n)$, where f/n is an operation of R, $x_1,\dots x_n$ are fresh variables, and the sorts of $f(x_1,\dots x_n)$ 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.

Example 6. *The following rule originates from instantiating* `y` *for "++" in the third rule of (4).*

```
H([]++(u++v)) = H(u++v)                                    (10)
```

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 $\mathbf{H}_f$. This operation is the composition of **H** with f, and then replaces every occurrence of the composition of **H** with f with $\mathbf{H}_f$.

Example 7. *The second phase transforms (10) into:*

```
H++([],u++v) = H++(u,v)                                    (11)
```

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 and the outcome of the first phase on C_R with C'_R. The mapping τ, from expressions of C_R to expressions of T_R, formally defines the transformation:

$$\tau(t) = \begin{cases} \mathbf{H}_f(\tau(t_1),\dots \tau(t_n)), & \text{if } t = \mathbf{H}(f(t_1,\dots t_n)); \\ s(\tau(t_1),\dots \tau(t_n)), & \text{if } t = s(t_1,\dots t_n), \text{ with } s \text{ symbol of } R; \\ v, & \text{if } t = v, \text{ with } v \text{ variable.} \end{cases} \tag{12}$$

T_R is more efficient than C_R because, for any operation f of R, the application of $\mathbf{H}_f$ avoids the allocation of a node labeled by f. This node is also likely to be pattern matched later.

Example 8. *Consider the usual length–of–a–list operation:*

```
length [] = 0
length (_:xs) = 1+length xs                                (13)
```

The compilation of (13), where we omit rules irrelevant to the point we are making, produces:

```
H(length([])) = 0
H(length(_:xs)) = H(1+length(xs))                          (14)
...
```

The transformation of (14), where again we omit rules irrelevant to the point we are making, produces:

```
Hlength([]) = 0
Hlength(_:xs)) = H+(1,length(xs))                          (15)
...
```

Below, we show the traces of a portion of the computations of `N(length [7])` *executed by* C_R *(left) and* T_R *(right), where the number* `7` *is an irrelevant value. The rules of "+" are not shown. Intuitively, they evaluate the arguments to numbers, and then perform the addition.*

$\mathbf{N}(\mathbf{H}(\texttt{length [7]})$	$\mathbf{N}(\mathbf{H}_{\texttt{length}}(\texttt{[7]})$
$\to \mathbf{N}(\mathbf{H}(\texttt{1}\underline{\texttt{+}}\texttt{length []})$	$\to \mathbf{N}(\mathbf{H}_{+}(\texttt{1,length []})$
$\to \mathbf{N}(\mathbf{H}(\texttt{1+}\mathbf{H}(\texttt{length []}))$	$\to \mathbf{N}(\mathbf{H}_{+}(\texttt{1,}\mathbf{H}_{\texttt{length}}(\texttt{[]})))$
$\to \mathbf{N}(\mathbf{H}(\texttt{1+0}))$	$\to \mathbf{N}(\mathbf{H}_{+}(\texttt{1,0}))$
$\to \mathbf{N}(\texttt{1})$	$\to \mathbf{N}(\texttt{1})$
$\to \texttt{1}$	$\to \texttt{1}$

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.

The transformation increases the size of a program. Certain rules are replaced by sets of rules. The number of rules in a replacing set is the number of constructors of some type. A coarse upper bound of the size of the transformed program is a constant factor of the size of the original program. Modern computers have gigabytes of memory. We believe that the growth in size could become a problem only in extreme cases, and likely would not be the most serious problem in those cases.

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. Every step of C_R is a step of C'_R and vice versa, modulo a renaming of nodes.*

Proof. Every rule of C'_R is an instance of a rule of C_R. Hence every step of C'_R is a step of C_R. For the converse, let $t \to u$ 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. Let v be the variable "wrapped" by $\mathbf{H}$ in r. Rule r is output by statement either 04 or 15 of procedure `compile`. We show that in both cases the match of v, say s, is an operation-rooted subexpression of t. 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 $r[f(x_1,\dots x_n)/v]$ is in C'_R. Therefore, modulo a renaming of nodes, $t \to u$ in C'_R $\square$

Corollary 3. *Let R be an inductively sequential system. For every operation-rooted expression t and head-constructor form u of R, $\mathbf{H}(t) \xrightarrow{+} u$ in C'_R if, and only if, $\tau(\mathbf{H}(t)) \xrightarrow{+} u$ in T_R modulo a renaming of nodes.*

Proof. Preliminarily, we show that for any s, $\mathbf{H}(t) \to s$ in C'_R iff $\tau(\mathbf{H}(t)) \to \tau(s)$ in T_R. Assume $\mathbf{H}(t) \to s$ in C'_R. There exists a rule $l \to r$ of C'_R and a match (graph homomorphism) σ such that $\mathbf{H}(t) = \sigma(l)$ and $s = \sigma(r)$. From the definition of phase 2 of the transformation, $\tau(l) \to \tau(r)$ is a rule of T_R. We show that this rule reduces $\tau(\mathbf{H}(t))$ to $\tau(s)$. Since τ is the identity on variables, and σ is the identity on non variables, $\sigma \circ \tau = \tau \circ \sigma$. Thus $\tau(\mathbf{H}(t)) = \tau(\sigma(l)) = \sigma(\tau(l)) \to \sigma(\tau(r)) = \tau(\sigma(r)) = \tau(s)$. The converse is similar because there a bijection between the steps of C'_R and T_R.

Now, we prove the main claim. First, the claim just proved holds also when $\mathbf{H}(t)$ is in a context. Then, an induction on the length of $\mathbf{H}(t) \xrightarrow{+} u$ in C'_R shows that $\tau(\mathbf{H}(t)) \xrightarrow{+} \tau(u)$ in T_R. Since by assumption u is an expression of R, by the definition of τ, $\tau(u) = u$. $\square$

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

Theorem 2. *Let R be an inductively sequential system. For all expressions t and u of R, $\mathbf{N}(t) \xrightarrow{+} u$ in C_R if, and only if, $\mathbf{N}(t) \xrightarrow{+} u$ in T_R.*

Proof. In the computation of $\mathbf{N}(t)$ in C_R, by the definition of τ, each computation of $\mathbf{H}(s)$ in C_R, for some expression s, is transformed into a computation of $\tau(\mathbf{H}(s))$ in T_R. By Lemma 5, the former ends in a head-constructor form of R. Hence, by Corollary 3, $\tau(\mathbf{H}(s))$ ends in the same head-constructor form of R. Thus, $\mathbf{N}(t) \xrightarrow{+} u$ in T_R produces the same result. The converse is similar. □

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 signature symbol. An arithmetic operation, such as addition, retrieves the integers labeling its argument nodes, adds them together, and allocates a new node labeled by the result of the addition.

Our first benchmark evaluates `length(`l_1`++`l_2`)`, where `length` is the operation defined in (13). 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. The ratio between the execution times of T_R and C_R varies with the implementation language, the order of execution of some instructions, and other code details that would seem irrelevant to the work being performed. Therefore, we measure quantities that are language and code independent. The tabular entries are in units per 10 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 million.

length (l_1++l_2)	C_R	T_R	O_R
rewrite steps	10	6	6
shortcut steps	0	4	4
node allocations	20	16	12
node matches	40	26	18

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 (14), both arguments of the addition in the right-hand side are needed. This information is known at compile-time, therefore the compiler can wrap an application of **H** around the right operand of "+" in the right-hand side of the rule.

$$\mathbf{H}(\texttt{length(\_:xs)}) \texttt{ = } \mathbf{H}(\texttt{1+}\mathbf{H}(\texttt{length(xs)})) \tag{16}$$

The composition of **H** with `length` is replaced by $\mathbf{H}_{\texttt{length}}$ during the second phase. The resulting rule is:

$$\mathbf{H}_{\texttt{length}}\texttt{(\_:xs)) = }\mathbf{H}_{+}\texttt{(1,}\mathbf{H}_{\texttt{length}}\texttt{(xs))} \tag{17}$$

Of course, there is no need to allocate a node for expression `1`, the first argument of the addition, every time rule (15) or (17) is applied. A single node can be shared by the entire computation. However, since the first argument of the application of $\mathbf{H}_{+}$ is constant, this application can be specialized or partially evaluated as follows:

$$\mathbf{H}_{\texttt{length}}\texttt{(\_:xs))} = \mathbf{H}_{\texttt{+1}}\texttt{(}\mathbf{H}_{\texttt{length}}\texttt{(xs))} \tag{18}$$

The application of rule (18) allocates no node of the contractum. In our benchmarks, we ignore any optimization that is not directly related to shortcutting. 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 decreases. The effectiveness of T_R to reduce node allocations or pattern matching with respect to C_R varies with the program and the computation.

Our second benchmark computes the n-th Fibonacci number for a relatively large value of n. The program we compile is:

```
fib 0 = 0
fib 1 = 1                                   (19)
fib n = fib (n-1) + fib (n-2)
```

To keep the example simple, we assume that pattern matching is performed by scanning the rules in textual order. Therefore, the last rule is applied only when the argument of `fib` is neither 0 nor 1.

fib (n)	C_R	T_R	O_R
rewrite steps	10	8	8
shortcut steps	0	2	2
node allocations	24	22	10
node matches	44	26	16

The tabular entries are in units per 10 rewrite 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 about 17.5 million. With respect to C_R, T_R avoids the construction of the root of the right-hand side of the third rule of (19). O_R transforms the right-hand side of this rule into:

$$\mathbf{H}_{+}\texttt{(}\mathbf{H}_{\texttt{fib}}\texttt{(}\mathbf{H}_{-}\texttt{(n,1)),}\mathbf{H}_{\texttt{fib}}\texttt{(}\mathbf{H}_{-}\texttt{(n,2)))} \tag{20}$$

since every node that is not labeled by the variable or the constants `1` and `2` is needed. In this benchmark, there is also no need to allocate a node for either `1` or `2` every time (20) is constructed/executed. With this further optimization, the step would allocate no new node for the contractum, and the relative gains of our approach would be even more striking.

8 Functional Logic Programming

Our work is motivated by the implementation of functional logic languages. The graph rewriting systems modeling functional logic programs are a superset of the inductively sequential ones. A minimal extension consists of a single binary operation, called *choice*, denoted by the infix symbol “?”. An expression x `?` y reduces non-deterministically to x or y. There are approaches [5, 6] for rewriting computations involving the *choice* operation that produce all the values of an expression without ever making a non-deterministic choice. These approaches are ideal candidates to host our compilation scheme.

Popular functional logic languages allow variables, called *extra variables*, which occur in the right-hand side of a rewrite rule, but not in the left-hand side. Computations with extra variables are executed by *narrowing* instead of rewriting. Narrowing simplifies encoding certain programming problems into programs [4]. Since our object code selects rules in textual order, and instantiates some variables of the rewrite system, narrowing with our object code is not straightforward. However, there is a technique [7] that transforms a rewrite system *with* extra variables into an equivalent system *without* extra variables. Loosely speaking, "equivalent", in this context, means a system with the same input/output relation. In conjunction with this technique, our compiler generates code suitable for narrowing computations.

9 Related Work

The redexes that we reduce are needed to obtain a constructor-rooted expression, therefoer they are closely related to the notion of *root-neededness* of [22]. However, we are interested only in normal forms that are constructor forms. In contrast to a computation according to [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 [10, 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 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.

Our compilation scheme has similaties with deforestation [24], but is complementary to it. Both anticipate rule application, to avoid the construction of 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. However, our expressions are operation-rooted whereas in deforestation they are constructor-rooted. These techniques can be used independently of each other and jointly in the same program.

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 [8]. 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.

10 Conclusion

Our work addresses rewriting computations for the implementation of functional logic languages. We presented two major results.

The first result is a compilation scheme for inductively sequential graph rewriting systems. The object code generated by our scheme has very desirable properties: it is simple consisting of only two functions that take arguments by value, it is theoretically efficient by only executing needed steps, and it is complete in that it produces the value, when it exists, of any expression. The two functions of the object code are easily generated from the signature of the rewrite system and a traversal of the definitional trees of its operations.

The second result is a transformation of the object code that shortcuts some rewrite steps. Shortcutting avoids partial or total construction of the contractum of a step by composing one function of the object code with one operation symbol of the rewrite system signature. This avoids the construction of a node and in some cases and its subsequent pattern matching. Benchmarks show that the savings in node allocation and matching can be substantial.

Future work will rigorously investigate the extension of our compilation technique to rewrite systems with the *choice* operation and extra variables, as discussed in Sect. 8, as well as systematic opportunities to shortcut needed steps in situations similar to that discussed 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 the University of Oregon. The second author wishes to thank Zena Ariola for hosting this visit. The authors wish to thank Olivier Danvy for insightful comments and the anonymous reviewers for their careful reviews.

References

[1] S. Antoy (1992): *Definitional Trees*. In H. Kirchner & G. Levi, editors: *Proceedings of the Third International Conference on Algebraic and Logic Programming*, Springer LNCS 632, Volterra, Italy, pp. 143–157. Available at `http://dx.doi.org/10.1007/bfb0013825`.

[2] S Antoy (1993): *Normalization by Leftmost Innermost Rewriting*. In: *Proceedings of the Third International Workshop on Conditional Term Rewriting Systems*, Springer-Verlag, London, UK, pp. 448–457. Available at `http://dx.doi.org/10.1007/3-540-56393-8_36`.

[3] S. Antoy (2005): *Evaluation Strategies for Functional Logic Programming*. *Journal of Symbolic Computation* 40(1), pp. 875–903. Available at `http://dx.doi.org/10.1016/j.jsc.2004.12.007`.

[4] S. Antoy (2010): *Programming with Narrowing*. *Journal of Symbolic Computation* 45(5), pp. 501–522. Available at `http://dx.doi.org/10.1016/j.jsc.2010.01.006`.

[5] S. Antoy (2011): *On the Correctness of Pull-Tabbing*. *TPLP* 11(4-5), pp. 713–730. Available at `http://dx.doi.org/10.1017/S1471068411000263`.

[6] S. Antoy, D. Brown & S. Chiang (2006): *Lazy Context Cloning for Non-deterministic Graph Rewriting*. In: *Proceedings of the 3rd International Workshop on Term Graph Rewriting, Termgraph'06*, Vienna, Austria, pp. 61–70. Available at `http://dx.doi.org/10.1016/j.entcs.2006.10.026`.

[7] S. Antoy & M. Hanus (2006): *Overlapping Rules and Logic Variables in Functional Logic Programs*. In: *Proceedings of the Twenty Second International Conference on Logic Programming*, Springer LNCS 4079, Seattle, WA, pp. 87–101. Available at `http://dx.doi.org/10.1007/11799573_9`.

[8] S. Antoy & A. Jost (2013): *Are needed redexes really needed?* In: *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, ACM, New York, NY, USA, pp. 61–71. Available at `http://doi.acm.org/10.1145/2505879.2505881`.

[9] M. Bezem, J. W. Klop & R. de Vrijer (eds.) (2003): *Term Rewriting Systems*. Cambridge University Press. Available at `http://dx.doi.org/10.1145/979743.979772`.

[10] G. L. Burn, S. L. Peyton Jones & J. D. Robson (1988): *The Spineless G-machine*. In: *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, ACM, pp. 244–258. Available at `http://doi.acm.org/10.1145/62678.62717`.

[11] R. Echahed & J. C. Janodet (1997): *On constructor-based graph rewriting systems*. Technical Report 985-I, IMAG. Available at `ftp://ftp.imag.fr/pub/labo-LEIBNIZ/OLD-archives/PMP/c-graph-rewriting.ps.gz`.

[12] W. Fokkink & J. van de Pol (1997): *Simulation as a correct transformation of rewrite systems*. In: *In Proceedings of 22nd Symposium on Mathematical Foundations of Computer Science, LNCS 1295*, Springer, pp. 249–258. Available at `http://dx.doi.org/10.1.1.41.8118`.

[13] J. C. González Moreno, F. J. López Fraguas, M. T. Hortalá González & M. Rodríguez Artalejo (1999): *An Approach to Declarative Programming Based on a Rewriting Logic*. *The Journal of Logic Programming* 40, pp. 47–87. Available at `http://dx.doi.org/10.1016/S0743-1066(98)10029-8`.

[14] M. Hanus (1994): *The Integration of Functions into Logic Programming: From Theory to Practice*. *Journal of Logic Programming* 19&20, pp. 583–628. Available at `http://dx.doi.org/10.1.1.226.8638`.

[15] M. Hanus (2013): *Functional Logic Programming: From Theory to Curry*. In: *Programming Logics - Essays in Memory of Harald Ganzinger*, Springer LNCS 7797, pp. 123–168. Available at `http://dx.doi.org/10.1007/978-3-642-37651-1_6`.

[16] M. Hanus, S. Lucas & A. Middeldorp (1998): *Strongly sequential and inductively sequential term rewriting systems*. *Information Processing Letters* 67(1), pp. 1–8. Available at `http://dx.doi.org/10.1016/S0020-0190(98)00016-7`.

[17] M. Hanus (ed.) (2012): *Curry: An Integrated Functional Logic Language (Vers. 0.8.3)*. Available at `http://www.curry-language.org`.

[18] G. Huet & J.-J. Lévy (1991): *Computations in orthogonal term rewriting systems*. In J.-L. Lassez & G. Plotkin, editors: *Computational logic: essays in honour of Alan Robinson*, MIT Press, Cambridge, MA. Part I, pp. 395–414 and Part II, pp. 415–443.

[19] J. F. T. Kamperman & H. R. Walters (1996): *Simulating TRSs by Minimal TRSs a Simple, Efficient, and Correct Compilation Technique*. Technical Report CS-R9605, CWI.

[20] R. Kieburtz (1985): *The G-machine: A fast, graph-reduction evaluator*. In: *Functional Programming Languages and Computer Architecture*, LNCS 201, Springer, pp. 400–413. Available at `http://dx.doi.org/10.1007/3-540-15975-4_50`.

[21] F. J. López-Fraguas, E. Martin-Martin, J. Rodríguez-Hortalá & J. Sánchez-Hernández (2014): *Rewriting and narrowing for constructor systems with call-time choice semantics*. *TPLP* 14(2), pp. 165–213. Available at `http://dx.doi.org/10.1017/S1471068412000373`.

[22] A. Middeldorp (1997): *Call by Need Computations to Root-stable Form*. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, ACM, New York, NY, USA, pp. 94–105. Available at `http://dx.doi.org/10.1145/263699.263711`.

[23] M. J. O'Donnell (1977): *Computing in Systems Described by Equations*. Springer LNCS 58. Available at `http://dx.doi.org/10.1007/3-540-08531-9`.

[24] Philip Wadler (1988): *Deforestation: Transforming Programs to Eliminate Trees*. *Theor. Comput. Sci.* 73(2), pp. 231–248. Available at `http://dx.doi.org/10.1016/0304-3975(90)90147-A`.