

Needed Computations Shortcutting Needed Steps

(Extended abstract)

Sergio Antoy

Computer Science Dept.
Portland State University
Oregon, U.S.A.
antoy@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 result of this compilation is another constructor-based graph rewriting system that is normalizing for the original system when using an innermost strategy. We then modify the resulting rewrite system in a way that avoids totally or partially the construction of the *contracta* of some needed steps of a computation. The resulting rewrite system can be easily implemented by eager functions in a variety of programming languages. 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 [8].

$$\begin{aligned} \text{loop} &= \text{loop} \\ \text{snd } (_, y) &= y \end{aligned} \tag{1}$$

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 [10]. 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.

2 Compilation

We define a constructor-based graph rewriting system R in the customary way [6]. An *expression* is a single-rooted, directed, acyclic graph [6, Def. 2]. R is *inductively sequential*, i.e., the rules of every defined operation of R are organized in a *definitional tree* [1]. The inductively sequential systems are the intersection [9] of the strongly sequential systems [10] and the constructor-based systems [13].

For simplicity and abstraction, we present the rewrite system, C_R , of R as a constructor-based graph rewriting system as well. C_R has only two operations called *head* and *norm*, denoted $\mathbf{H}$ and $\mathbf{N}$, respectively. The constructors symbols of C_R are all and only the symbols of R .

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 top to bottom priority as in common functional languages. The rules of $\mathbf{H}$ contributed by an operation with definitional tree $\mathcal{T}$ are generated by the procedure `compile` defined below. 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.

```

compile  $\mathcal{T}$ 
01  case  $\mathcal{T}$  of
02    when  $\text{branch}(\pi, o, \tilde{\mathcal{T}})$  then
03       $\forall \mathcal{T}_i \in \tilde{\mathcal{T}}$  compile  $\mathcal{T}_i$ 
04      output " $\mathbf{H}(\{\pi\}) = \mathbf{H}(\{\pi[\mathbf{H}(\pi|_o)]_o\})$ "
05    when  $\text{rule}(\pi, l \rightarrow r)$  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  $l' \rightarrow r' = (l \rightarrow r)[c(x_1, \dots, x_n)/r]$ 
14            output " $\mathbf{H}(\{l'\}) = \{r'\}$ "
15            output " $\mathbf{H}(\{l\}) = \mathbf{H}(\{r\})$ "
16      when  $\text{exempt}(\pi)$  then
17        output " $\mathbf{H}(\{\pi\}) = \text{abort}$ "

```

(2)

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 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.

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

```

[] ++ y = y
(x : xs) ++ y = x : (xs ++ y)

```

(3)

The definitional tree of "++", see [3, p. 14], consists of a single *branch* where the first argument of "++"

is the inductive variable and the children are the two *rules*. Applying procedure `compile` to this tree produces the following output:

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

Operation $\mathbf{N}$ of the rewrite system is defined by one rule for each symbol of R . In the following *metarules*, c stands for a constructor or arity m of R , f stands for an operation or arity n of R , and x_i is a fresh variable for every i .

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

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

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

The rewrite system C_R consists of the $\mathbf{H}$ rules generated by procedure `compile` for the operations of R and the $\mathbf{N}$ rules generated according to (5) for all the symbols of R .

Let $\mathcal{E}$ be the function that takes an expression t of C_R and “erases” from t every occurrence of $\mathbf{N}$ and $\mathbf{H}$ thus producing an expression of R . Let “ $\bar{\rightarrow}$ ” denote the reflexive closure of “ $\rightarrow$ ”.

Theorem 1. *Let R be an inductively sequential graph rewriting system. For all expressions t of R , let $\mathbf{N}(t) = t_0 \rightarrow t_1 \rightarrow \dots$ be an innermost computation in C_R . Then, $\mathcal{E}(t_0) \bar{\rightarrow} \mathcal{E}(t_1) \bar{\rightarrow} \dots$ is a needed computation in R .*

Theorem 2. *Let R be an inductively sequential graph rewriting system. For all expressions t and constructor expressions u of R , $t \xrightarrow{*} u$ in R iff $\mathbf{N}(t) \xrightarrow{+} u$ in C_R modulo a renaming of nodes.*

3 Transformation

We transform the rewrite system C_R to avoid totally or partially the construction of certain contracta. The transformation consists of two phases. The first phase instantiates each variable of C_R that occurs as the argument of a recursive invocation of $\mathbf{H}$. Examples of such variables can be seen in (4) such as x in the fifth rule and y in the third rule. We instantiate any such variable with $f(x_1, \dots, x_n)$, where f ranges over all the operations of R , n is the arity of f , and every x_i is a fresh variable. For example, the following rule originates from instantiating y for operation $++$ in the third rule of (4).

$$\mathbf{H}([\] ++ (u ++ v)) = \mathbf{H}(u ++ v) \tag{7}$$

There is a similar rule for each operation of R .

The first phase of the transformation ensures that $\mathbf{H}$ 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 $\mathbf{H}_f$, which is the composition of $\mathbf{H}$ with f . For example, (7) becomes:

$$\mathbf{H}_{++}([\], u ++ v) = \mathbf{H}_{++}(u, v) \tag{8}$$

After the second phase, the operation $\mathbf{H}$ can be eliminated from the rewrite system, since it is no longer invoked. We denote the transformed C_R with T_R . The key result of C_R is preserved by T_R .

Theorem 3. *Let R be an inductively sequential graph rewriting system. For all expressions t and constructor expressions u of R , $t \xrightarrow{*} u$ in R iff $\mathbf{N}(t) \xrightarrow{*} u$ in T_R modulo a renaming of nodes.*

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 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 $\mathbf{N}(([] ++ []) ++ t)$ executed by C_R (left) and T_R (right), where t is an irrelevant expression.

$\mathbf{N}(\mathbf{H}(([] ++ []) ++ t))$	$\mathbf{N}(\mathbf{H}_{++}([] ++ [], t))$
$\mathbf{N}(\mathbf{H}(\mathbf{H}([] ++ []) ++ t))$	$\mathbf{N}(\mathbf{H}_{++}(\mathbf{H}_{++}([], []), t))$
$\mathbf{N}(\mathbf{H}(\underline{[] ++ t}))$	$\mathbf{N}(\mathbf{H}_{++}([], t))$
$\mathbf{N}(t)$	$\mathbf{N}(t)$

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 .

4 Benchmarking

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 computation evaluates $\text{length}(l_1 ++ l_2)$, where length is the usual length-of-a-list operation and l_1 and l_2 are lists of one million elements. The tabular entries are again in millions plus or minus a few units.

$\text{length}(l_1 ++ l_2)$	C_R	T_R
rewrite steps	5	3
shortcut steps	0	2
node allocations	10	8
node matches	20	13

5 Related Work and Conclusion

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 [12].

A compilation scheme similar to ours is described in [2]. The work is about term rather than graph rewriting, and 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 [7, 11]. These efforts are more operational than ours. A compilation scheme similar to ours is described in [5]. The compilation scheme is different. It does not claim to execute only needed steps, but 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.

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 and avoid the application of overlapping rules [4] without affecting the results of computations. Future work will exploit further opportunities, not discussed here but already available in the simple benchmark of Sect. 4, to shortcut more needed steps.

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

- [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, D. Brown & S. Chiang (2006): *Lazy Context Cloning for Non-deterministic Graph Rewriting*. In: *Proc. 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>.
- [5] 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>.
- [6] R. Echahed (2008): *Inductively Sequential Term-Graph Rewrite Systems*. In: *Graph Transformations, 4th International Conference (ICGT 2008)*, Springer, LNCS 5214, Leicester, UK, pp. 84–98. Available at http://dx.doi.org/10.1007/978-3-540-87405-8_7.
- [7] W. Fokkink & J. van de Pol (1997): *Simulation as a correct transformation of rewrite systems*. 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>.
- [8] M. Hanus, editor (2006): *Curry: An Integrated Functional Logic Language (Vers. 0.8.2)*. Available at <http://www-ps.informatik.uni-kiel.de/currywiki/>.
- [9] 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](http://dx.doi.org/10.1016/S0020-0190(98)00016-7).
- [10] 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.
- [11] 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.
- [12] 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, doi:10.1145/263699.263711. Available at <http://doi.acm.org/10.1145/263699.263711>.
- [13] 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>.