A General Natural Rewriting Strategy

Santiago Escobar[1], José Meseguer[2], and Prasanna Thati[3]

[1] Universidad Politécnica de Valencia, Spain
sescobar@dsic.upv.es
[2] University of Illinois at Urbana-Champaign, USA
meseguer@cs.uiuc.edu
[3] Carnegie Mellon University, USA
thati@cs.cmu.edu

Abstract. We define an efficient rewriting strategy for general term rewriting systems. Several strategies have been proposed over the last two decades for rewriting, the most efficient of all being the *natural rewriting strategy* of Escobar. All the strategies so far, including natural rewriting, assume that the given term rewriting system is left-linear and constructor-based. Although these restrictions are reasonable for some functional programming languages, they limit the expressive power of equational programming languages, and they preclude certain applications of rewriting to equational theorem proving and to languages combining equational and logic programming. In [5], we proposed a conservative generalization of the natural rewriting strategy that does not require the previous assumptions for the rules and we established its soundness and completeness.

1 Introduction

A challenging problem in modern programming languages is the discovery of sound and complete evaluation strategies which are: (i) optimal w.r.t. some efficiency criterion, (ii) easily implementable, and (iii) applicable for a large class of programs. This was first addressed in a seminal paper by Huet and Levy [10], where the *strongly needed reduction* strategy was proposed. Several refinements of this strategy have been proposed over the last two decades, the most significant ones being Sekar and Ramakrishnan's *parallel needed reduction* [11], and Antoy, Echahed and Hanus' *(weakly) outermost-needed rewriting* [1]. Recently, (weakly) outermost-needed rewriting has been improved by Escobar by means of the *natural rewriting strategy* [6,7]. Natural rewriting is based on a suitable refinement of the demandedness notion associated to (weakly) outermost-needed rewriting.

However, a typical assumption of the above rewriting strategies, including the natural rewriting strategy, is that the rewrite rules are left-linear and constructor-based. These restrictions are reasonable for some functional programming languages, but they limit the expressive power of equational languages such as OBJ [9], CafeOBJ [8], ASF+SDF [4], and Maude [3], where non-linear

left-hand sides are perfectly acceptable. This extra generality is also necessary for applications of rewriting to equational theorem proving, and to languages combining equational and logic programming, since in both cases assuming left-linearity is too restrictive. Furthermore, for rewrite systems whose semantics is not equational but is instead rewriting logic based, such as rewrite rules in ELAN [2], or Maude system modules, the constructor-based assumption is unreasonable and almost never holds.

In summary, generalizing natural rewriting to *general* rewriting systems will extend the scope of applicability of the strategy to more expressive equational languages and to rewriting logic based languages, and will open up a much wider range of applications. In the following, we give the reader an intuitive example of how the generalized natural rewriting strategy works.

Example 1. Consider the following TRS for proving equality of arithmetic expressions built using division ($\div$), modulus or remainder ($\%$), and subtraction ($-$) operations on natural numbers.

$$\begin{array}{ll}
\texttt{(1) } \texttt{0} \div \texttt{s(N)} \to \texttt{0} & \texttt{(5) } \texttt{M} - \texttt{0} \to \texttt{M} \\
\texttt{(2) } \texttt{s(M)} \div \texttt{s(N)} \to \texttt{s((M}-\texttt{N)} \div \texttt{s(N))} & \texttt{(6) } \texttt{s(M)} - \texttt{s(N)} \to \texttt{M}-\texttt{N} \\
\texttt{(3) } \texttt{M} \,\%\, \texttt{s(N)} \to \texttt{(M}-\texttt{s(N))} \,\%\, \texttt{s(N)} & \texttt{(7) } \texttt{X} \approx \texttt{X} \to \texttt{True} \\
\texttt{(4) } \texttt{(0} - \texttt{s(M))} \,\%\, \texttt{s(N)} \to \texttt{N} - \texttt{M} &
\end{array}$$

Note that this TRS is not left-linear because of rule (7) and it is not constructor-based because of rule (4). Therefore, it is outside the scope of all the strategies mentioned above. Furthermore, note that the TRS is neither terminating nor confluent due to rule (3).

Consider the term $t_1 = \texttt{10!} \div \texttt{0}$. If we only had rules (1), (2), (5) and (6), the natural rewriting strategy of [6] would be applicable and no reductions on t_1 would be performed, since t_1 is a head-normal form. In contrast, other strategies such as outermost-needed rewriting [1] would force[4] the evaluation of the computationally expensive subterm `10!`. Hence, we would like to generalize natural rewriting to a version that enjoys this optimality and that can also handle non-left-linear and non-constructor-based rules such as (7) and (4).

Further, consider the term $t_2 = \texttt{10!} \,\%\, \texttt{(1}-\texttt{1)} \approx \texttt{10!} \,\%\, \texttt{0}$. We would like the generalized natural rewriting strategy to perform only the optimal computation:

$$\begin{array}{l}
\texttt{10!} \,\%\, \underline{\texttt{(s(0)}-\texttt{s(0))}} \approx \texttt{10!} \,\%\, \texttt{0} \to \texttt{10!} \,\%\, \underline{\texttt{(0}-\texttt{0)}} \approx \texttt{10!} \,\%\, \texttt{0} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \underline{\texttt{10!} \,\%\, \texttt{0} \approx \texttt{10!} \,\%\, \texttt{0}} \to \texttt{True}
\end{array}$$

that avoids unnecessary reduction of the subterm $\texttt{10!} \,\%\, \texttt{0}$ at the final rewrite step and avoids also reductions on any term `10!`.

In [5], we proposed a conservative generalization of the demandedness notion of [6,7] that drops the assumptions that the rewrite rules are left-linear and constructor-based, while retaining soundness and completeness w.r.t. head-normal forms. In the following, we provide some useful insights about this generalization and refer the reader to [5] for further details.

[4] See [6, Example 21] for a formal justification of this fact.

2 Generalized Natural Rewriting

We are interested in a lazy strategy that, to the extent possible, performs only those reductions that are essential for reaching head-normal forms, which is consistent with a lazy behavior of functional programs. We adopt the approach of computing a *demanded* set of redexes in t such that *at least* one of the redexes in the demanded set has to be reduced before *any* rule can be applied at the root position in t; this is a common idea in lazy evaluation strategies for programming languages.

The basic approach is that if a term t is not a head-normal form, then we know that after a (possibly empty) sequence of rewrites at positions below the root, a rule $l \to r$ can be applied at the root. Thus, we compare the term t with each left-hand side l in the TRS and obtain the *least general context* between t and l, i.e., the maximal common parts between the terms t and l, in order to bound what we called *disagreeing positions* between t and l. In [5], we defined an operator $\mathcal{P}os_{\neq}(T)$ for a set of terms T that returns the set of disagreeing positions between all the terms in T. Also, we defined an operator $DP_l(t)$ that returns the set of disagreeing positions between t and l and which uses $\mathcal{P}os_{\neq}(T)$ in order to deal with non-linear variables in l.

Example 2. Consider the left-hand side $l_7 = \mathtt{X} \approx \mathtt{X}$ of the rule (7) and the term $t_2 = \mathtt{10!} \ \% \ \mathtt{(1-1)} \approx \mathtt{10!} \ \% \ \mathtt{0}$ of Example 1. The least general context of l_7 and t_2 is $s = \mathtt{W} \approx \mathtt{Y}$. Now, while computing $DP_{l_7}(t_2)$, we obtain the set of disagreeing positions between the subterms in t_2 corresponding to the non-linear variable $\mathtt{X}$ in l_7, i.e. the set $\mathcal{P}os_{\neq}(\mathtt{10!} \ \% \ \mathtt{(1-1)}, \mathtt{10!} \ \% \ \mathtt{0}) = \{2\}$. And thus we conclude that $DP_{l_7}(t_2) = \{1, 2\}.\{2\} = \{1.2, 2.2\}$.

Moreover, in [5], we filtered some disagreeing positions using the notion that t is *failing w.r.t.* the left-hand side l, denoted by $l \blacktriangleleft t$. We say t is failing w.r.t. l if there is no rewrite sequence starting from t such that the first reduction in the sequence at the root position uses the rule $l \to r$. This notion is ultimately related to the notion of constructor clash between a term and a left-hand side, which is also common in lazy evaluation strategies for programming languages.

Example 3. Consider the terms $t = \mathtt{10!} \ \% \ \mathtt{0}$ and $l_3 = \mathtt{M} \ \% \ \mathtt{s(N)}$ from Example 1. We have that $l_3 \blacktriangleleft t$ because the position $2 \in DP_{l_3}(t)$ has two different symbols at l_3 and t and no possible reduction can make them equal; similarly $l_4 \blacktriangleleft t$. Now, consider the terms $t' = \mathtt{s(Z)} \approx \mathtt{0}$ and $l_7 = \mathtt{X} \approx \mathtt{X}$, again from Example 1. In this case, which considers non-linear variables in l_7, we have $l_7 \blacktriangleleft t'$, since no reduction can make positions 1 and 2 in $DP_{l_7}(t')$ equal.

Then, in [5], we collected all the demandedness information obtained from the left-hand sides that are not failing and built a set of *demanded redexes*, denoted by $DR(t)$, which recursively calls itself on the demanded positions obtained by each $DP_l(t)$, for all l in the TRS, and their positions above them. However, a further refinement was included in [5], since not all the demanded positions are inspected by $DR(t)$, but only those positions which are the *most frequently demanded positions* and which cover all the rules in the TRS.

Example 4. Continuing all the previous Examples, consider the computation of the set $DR(t_2)$. We have that $DP_{l_7}(t_2) = \{1.2, 2.2\}$ for the left-hand side l_7 of the rule (7). Then $DR(t_2)$ calls itself recursively on positions 1.2, 2.2, and their positions above them, i.e., positions 1 and 2. Position 2.2 is rooted by a constructor symbol and no rule is applicable, so $DR(t_2|_{2.2}) = \varnothing$. By Example 3, subterm at position 2 is failing w.r.t. the TRS, so $DR(t_2|_2) = \varnothing$. Position 1.2 is already a redex, so we say $DR(t_2|_{1.2}) = \{\langle \Lambda, (6)\rangle\}$. Now, consider the subterm $t_2|_1 =$ `10! % (1-1)` and the left-hand sides of the rules (3) and (4). We have that $DP_{l_3}(t_2|_1) = \{2\}$ and $DP_{l_4}(t_2|_1) = \{1, 2\}$. However, the set $P = \{2\}$ covers $DP_{l_3}(t)$ and $DP_{l_4}(t)$ and then, position 2 is considered as the most frequently demanded position for $t_2|_1$ in the TRS. Then, $DR(t_2|_1)$ will call recursively to $DR(t_2|_{1.2})$, which was computed before, and finally we conclude $DR(t_2|_1) = \{\langle 2, (6)\rangle\}$ and, obviously, $DR(t_2) = \{\langle 1.2, (6)\rangle\}$. That is, we have avoided any unnecessary reduction, following the optimal sequence of Example 1.

References

1. S. Antoy. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming ALP'92*, volume 632 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, Berlin, 1992.
2. P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
4. A. van Deursen, J. Heering, and P. Klint. *Language Prototyping: An Algebraic Specification Approach*. World Scientific, 1996.
5. S. Escobar, J. Meseguer, and P. Thati. Natural rewriting for general term rewriting systems. In *Pre-proceedings of 14th International Workshop on Logic-based Program Synthesis and Transformation, LOPSTR'04*, 2004. To appear.
6. S. Escobar. Refining weakly outermost-needed rewriting and narrowing. In D. Miller, editor, *Proc. of 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'03*, pages 113–123. ACM Press, New York, 2003.
7. S. Escobar. Implementing natural rewriting and narrowing efficiently. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, volume 2998 of *Lecture Notes in Computer Science*, pages 147–162. Springer-Verlag, Berlin, 2004.
8. K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
9. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.
10. G. Huet and J.-J. Lévy. Computations in Orthogonal Term Rewriting Systems, Part I + II. In *Computational logic: Essays in honour of J. Alan Robinson*, pages 395–414 and 415–443. The MIT Press, Cambridge, MA, 1992.
11. R.C. Sekar and I.V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. *Information and Computation*, 104(1):78–109, 1993.