11th Int. Workshop on Functional and (constraint) Logic Programming (WFLP'02)
Grado, Italy — June 20–22, 2002.
Research Report UDMI/18/2002/RR, pages 35–43

Demandness in Rewriting and Narrowing

Sergio Antoy[1]* and Salvador Lucas[2]**

[1] Computer Science Department
Portland State University
Portland, OR 97207, U.S.A.
antoy@cs.pdx.edu

[2] DSIC
Universidad Politécnica de Valencia
Spain
slucas@dsic.upv.es

Abstract. The traditional investigation of rewriting and narrowing strategies aims at establishing fundamental properties, such as soundness, completeness and/or optimality, of a strategy. In this work, we analyze and compare rewriting and narrowing strategies from the point of view of the *information* taken into account by a strategy to compute a step. The notion of *demandness* provides a suitable framework for presenting and comparing well-known strategies. We find the existence of an almost linear sequence of strategies that take into account more and more information. We show on examples that, as we progress on this sequence, a strategy becomes more focused and avoids some useless steps computed by strategies preceding it in this sequence. Our work, which is still in progress, clarifies the behavior of similar or related strategies and it promises to simplify the transfer of some results from one strategy to another. It also suggests that the notion of demandness is both atomic and fundamental to the study of strategies.

1 Introduction

Modern functional logic programs are, for the most part, modeled by constructor based term rewriting systems and are executed by narrowing. Narrowing is a generalization of rewriting that allows the evaluation of expressions possibly containing incomplete information, which is represented by uninstantiated variables.

An essential component of the execution of functional logic programs is a narrowing strategy, i.e., the policy or algorithm that determines both which subexpression of an expression should be evaluated first and the binding, if any, of uninstantiated variables. Over the years, many strategies have been proposed.

* Supported in part by the NSF under grants INT-9981317 and CCR-0110496
** Work partially supported by CICYT TIC2001-2705-C03-01, Acciones Integradas HI 2000-0161, HA 2001-0059, HU 2001-0019, and Generalitat Valenciana GV01-424.

Some of these generalize to narrowing well-known behaviors of functional evaluation such as call-by-value or call-by-need. Some of these strategies are fairly complicated. Other strategies propose a simpler approach where the evaluation of some arguments of a function call is prioritized by means of replacement maps, which are often defined by the programmer. For example, a map may establish which arguments of a function call should always (or never) be evaluated before the call.

A significant difference between functional programming and functional logic programming is that, in some cases, the latter is modeled by larger classes of rewrite systems, in particular, rewrite systems that support non-deterministic computations. Non-determinism originates from choices of overlapping rewrite rules. With this kind of rules, narrowing strategies become more complicated. The classic notions of call-by-value and call-by-need are no longer applicable or meaningful and the properties of the proposed strategies are less well understood.

A non-negligible number of strategies can be informally classified as *demand driven*. Some demand driven strategies are applicable to classes of rewrite systems much larger than those modeling first order computations in typical functional languages. *Demand driven* strategies are informally characterized as follows. If possible, a term t is evaluated (reduced or narrowed) at the top. Otherwise, some arguments of the root of t are (recursively) evaluated if they might promote the application of a rewrite rule at the top—in other words, if a rule "demands" the evaluation of these arguments.

This notion of "demandness," which varies among strategies and will be specialized later, is simple and often practical, but imperfect in some situations. The fact that a rule demands the evaluation of certain arguments, neither implies that the evaluation of those arguments guarantees the application of the rule nor that the application of the rule is necessary, in some intuitive sense, to the (full) evaluation of t. In general, these problems are undecidable. In the worst case, a demand driven strategy may fail to terminate a computation even when termination is possible (incompleteness). In other cases, the strategy may simply execute steps that a smarter strategy would avoid (wastefulness or non-optimality). Finally, a demand driven strategy may terminate a computation, but without yielding a normal form (incorrectness).

In this paper, we recall a few rewriting or narrowing strategies, we characterize and analyze them from the viewpoint of "demandness" as discussed earlier, and we specifically compare what each strategy is looking at to compute a step. The main result of our investigation is that the more information a strategy is looking at the more focused the strategy is. More precisely, a strategy that look at more information is able to execute fewer steps to compute a result. This conclusion is hardly surprising. However, our novel approach allows us to study within the same framework both simple strategies, such as *context sensitive rewriting strategies* [12], and more complex strategies such as *needed narrowing* [7]. Our approach makes it easier to understand both each strategy in isolation and some relationships between strategies. This eases the choices of

which strategy is more appropriate to use for certain applications or in certain contexts.

Section 2 briefly recalls some strategies and presents them from the viewpoint of demandness. Section 3 offers our conclusion.

2 Strategies

Intuitively, a demand driven strategy works as follows. Given a term t to evaluate, the strategy aims at applying every possible rewrite rule at the root of t. Let $l \to r$ be a rule. In some cases, e.g., when t and l unify, $l \to r$ is immediately applicable. In other cases, e.g., when the leading symbol of l and the root of t differ, one can immediately exclude the rule. The interesting cases are those in which it is not immediate to say whether a rule might eventually be applicable. This occurs, e.g., when the roots of t and l are the same, but the subterm of l at some position p and the corresponding subterm $t|_p$ of t do not unify. Since we consider constructor based term rewriting systems, the subterm of l at position p is constructor-rooted. If the subterm of t at position p is also constructor-rooted, then we can immediately exclude the rule. If the subterm of t at position p is operation-rooted, it needs to be evaluated before the rule $l \to r$ can be applied at the root of (a descendant of) t. Loosely speaking, in this case the evaluation of $t|_p$ is *demanded* by $l \to r$. Different demand driven strategies go about selecting p in different ways.

For example, consider the following familiar operations on lists. The notation adopts the syntax of Curry [9], which in this case is identical to Haskell [18], except that we represent natural numbers in Peano notation. This choice both simplifies the code and keeps a closer correspondence between the program and the rewrite system modeling it.

```
append [] ys = ys
append (x:xs) ys = x:append xs ys

drop 0 xs = xs                                        (1)
drop (Succ _) [] = []
drop (Succ n) (_:xs) = drop n xs
```

Given the term $t =$ `(drop (n+m) (append p q))`, some demand driven strategies would evaluate either `(n+m)` or `(append p q)` or both in hopes that once these subterms are evaluated some rule of `drop` could become applicable to the resulting term.

This example prompts several considerations. (C1) the arguments of `drop` are not all alike. Without looking at the right-hand sides, it would seem preferable to evaluate `(n+m)` than `(append p q)`. The reason is that depending on the value of `(n+m)`, the evaluation of `(append p q)` might not be needed. More precisely, if `(n+m)` does not evaluate to a constructor-rooted term, t itself cannot be evaluated to a constructor-rooted term. Since in functional (logic) languages normal forms are interesting only if they are constructor terms, in this example the evaluation of `(append p q)` would be useless. Many demand driven strategies are not that

sophisticated, but for some classes of rewrite systems for which a demand driven strategy is intended, this sophistication may be impossible, e.g., see the definition of operation `insert` in Display (2). (C2) some demand driven strategies use individual rules for selecting which term to evaluate. This condition leads to suboptimal computations, in some cases. (C3) a demand driven strategy is top-down. By this, we mean that a demand driven strategy selects which subterm of a term t to evaluate by traversing t from the root down to the leaves. This top-down traversal has two interrelated and somewhat subtle consequences: (C3a) a demand driven strategy is mostly lazy in the sense that the arguments of a function are preferably not evaluated, but the strategy is not completely lazy in the sense that only unavoidable steps are performed, see (1), and (C3b) a demand driven strategy is mostly outermost in the sense that inner narrexes of a term are preferably not narrowed, but the strategy is not completely outermost in the sense that only outermost narrexes are narrowed, see (2).

The following example defines an operation that inserts an element in a list at some position non-deterministically chosen. This operation is typically used, e.g., for computing the permutations of a list or for extracting some element from a list.

```
insert x y = x:y
insert x (y:ys) = y:insert x ys
```
(2)

The expression `(insert 0 [1,2])` yields any of `[0,1,2]`, `[1,0,2]` or `[1,2,0]`. Now, consider the evaluation of $t =$ `(insert 0 (append p q))`. Among the computations of t, one evaluates `(append p q)`, but another does not. There is no universally accepted notion of needed step for computations involving operations of this kind, and the computation that evaluates `(append p q)` is not outermost.

In the remainder of this section, we recall some strategies and present them from the viewpoint of demandness.

Context-sensitive rewriting strategies [12]

Context-sensitive rewriting (CSR) uses a *replacement map* defining which indexes of the arguments of a function (call) to evaluate [13]. Any evaluation is forbidden for the other arguments. The definition of the map is the responsibility of the programmer. As a rule of thumb, the arguments of a function call to evaluate are those corresponding to non-variable terms in the left-hand side of some rewrite rule. The map obtained in this way is called *canonical*. The canonical replacement map ensures that head-normal forms can be computed for left-linear rewrite systems [13]. For example, the canonical map of operation `append` defined in (1) contains position 1, but not position 2. The reason is that unless the first argument of `append` is evaluated, no rule of `append` can be fired. The value of the second argument of `append` does not affect the application of either rule. The definition of canonical replacement map is akin to the notion of demandness (by the rules of the term rewriting system), see Section 5.1 of [13].

The canonical map of operation `drop` defined in (1) contains both positions 1 and 2. This strategy look at each argument of each rewrite rule individually. Therefore, the strategy is unable to determine that the evaluation of the second argument of `drop` might not be needed depending on the value of the first argument.

Lazy rewriting strategies [8,15]

Lazy rewriting (LR) uses a *replacement map* similarly to CSR. This map is intended to define a 'pure' eager behavior (since the 'lazy' or demanded behavior is achieved in a different manner). Again, the replacement map of LR can be defined by the programmer. As a rule of thumb, given a defined function f, the map contains the arguments of f that are non-variable subterms in *all* the rules defining[3] f (for instance, $\mu(\texttt{drop}) = \{1\}$ for `drop` as defined in (1)). The evaluation of such subterms is intended to take place before the evaluation of the other subterms. Then, the (other) operation rooted subterms of a function call are evaluated if both they correspond to non-variable terms in the left-hand side of some rewrite rule R defining the called function and the symbols occurring in the path from this position to the root of the rule coincide in R. Additionally, it is required that the already evaluated part of the term matches the corresponding part of the left-hand side of the rule. Subterms so evaluated are referred to as 'activated' in the terminology of [8,15]. Hence, the activation and evaluation of such subterms is intended to (eventually) take place after the evaluation of subterms indicated by the replacement map. Finally, we note that LR with the canonical map coincides with CSR [15].
In particular, the evaluation of $t =$ `(drop (n+m) (append p q))` proceeds by evaluating `(n+m)`. If `(n+m)` does not evaluate to 0, then LR activates the subterm of t at position 2; otherwise, this subterm is not activated. Note that the evaluation of the second argument of the resulting term, namely `(drop 0 (append p q))`, is not demanded. Thus, LR performs better than CSR because it looks at some context of the arguments that are candidates for evaluations. But note that active parts of a recently activated subterm can *automatically* become active within the whole term. Hence, they can be freely evaluated *even without being demanded by any rule*. Thus, lazy rewriting is not so lazy, after all.

On-demand rewriting strategies [14]

On-demand rewriting (ODR) uses two *replacement maps*, μ and μ_D, for each symbol of the signature. The map μ is as in CSR. The map μ_D relaxes the requirement that any evaluation is forbidden for arguments not in the map μ. For those arguments, the evaluation is allowed 'on-demand', according to

[3] Strictness information can also be useful for defining the replacement map.

the left-hand side of a rule. The definition of the maps is the responsibility of the programmer. As a rule of thumb, selecting the first replacement map as for LR, and the canonical replacement map for the second replacement map would ensure that head-normal forms are computed for left-linear constructor based rewrite systems [14] and the more accurate comparison with LR.

Demand driven rewriting and narrowing [1,2,11,16]

Several strategies have appeared in the literature under the generic name of *demand driven*. A common characteristic of these strategies is the use of sets of demanded positions instead of replacement maps. A set of demanded positions defines which arguments of a function (call) to evaluate. In principle, this is similar to a replacement map, but different occurrences of a same function may have different sets of demanded positions. In an intuitive sense, an argument of a function call may or may not be evaluated depending on the values of other arguments. A variant of this strategy [2], proposed for non-deterministic functional computations in logic programming, uses definitional trees and is therefore limited to constructor based rewrite systems. [11] generalizes this strategy to narrowing computations, but restrict it to conditional weakly orthogonal rewrite systems.
For example, second argument of a call to the function `drop` defined in (1) is evaluated only when the first argument is `Succ`-rooted. It is not evaluated in calls where the first argument is `0`. DDR is more sophisticated than CSR, ODR, and LR in some cases, and without burdening the programmer with the definition of a replacement map. [2] applied to a term rooted by operation `insert` defined in (2) non-deterministically either applies the first rule without evaluating any argument or it evaluates the second argument, if it is not already constructor-rooted, to attempt to apply the second rule.

Needed rewriting and narrowing [10,7,4]

(Strongly) Needed rewriting (NR) simultaneously looks at the left-hand sides of *all* the rules defining an operation. Loosely speaking, an argument of a function call is evaluated only when it is demanded by all the rules that could be applied to that call. In other words, some arguments are more popular than others and the strategy evaluates a position according to this popularity order. In strongly sequential systems, the order of evaluation is determined by matching automata [10]. In inductively sequential, the order of evaluation is determined by definitional trees [3]. This strategy extends the behavior of call-by-need computations to narrowing. [4] extends this strategy to rewrite systems with a particular kind of overlapping rules. All these strategies are quite accurate in the sense that they compute only necessary steps—[4] modulo non-deterministic choices. The counterside of this accuracy is a limitation in the kinds of rewrite systems to which these strategies

are applicable. These strategies have been proved to be complete and optimal for the domains for which they are intended.
For example, none of these strategies is applicable to the operation `insert` defined in (2). Applied to operation `drop` defined in (1) these strategies first evaluate the first argument and then evaluate the second argument if the first is `Succ`-rooted.
It is interesting to observe that these strategies are not explicitly defined in term of demandness or replacement maps, but are closely related to the previous strategies (see [12] for a detailed comparison). One may obtain an *individual* canonical map for each rule defining an operation. CSR obtains the canonical map of an operation as the union of all these individual maps. NR finds needed positions as the intersection of all the individual maps applicable to a function call.

Weakly needed rewriting and narrowing [19,6].

WNR, similar to NR, simultaneously looks at all the rules of a single operation. By continuing the analogy introduced for NR, WNR, similar to CSR, obtains the canonical map of an operation as the union of all the individual maps of each rule. However, by contrast to CSR, WNR evaluates in parallel all the demanded positions of all the rules potentially applicable to a function call. WNR is a relatively simple strategy for rewriting [19]. For narrowing, the situation becomes much more complicated because the steps at distinct positions may require incompatible substitutions and consequently cannot be executed in parallel as for rewriting. A narrowing strategy that conservatively extends [19] is presented in [6]. Both strategies have been proved to be complete for the constructor based weakly orthogonal rewrite systems. However, the optimality result of the former have not been entirely proved for the latter.
These strategies are intended for weakly orthogonal rewrite systems. In these, there exist terms without needed positions. This may occur when some rewrite rules overlap as in the following well-known *parallel-or* operations:

```
or True _ = True
or _ True = True                                    (3)
or False False = False
```

For the sake of completeness, we also mention a couple of important strategies that are not based on demandness. The parallel outermost and outermost fair strategies [17] have been investigated for rewriting only. They radically differ from the previous ones in that they do not look specifically at the rewrite rules. Rather they look at the outermost redexes of term. They are complete for the same class for which WNR is complete. It is easy to see on simple examples that these strategies may evaluate more redexes than WNR does.

Another strategy not directly based on demandness is defined in [5]. This strategy is important because it is the first complete narrowing strategy for the

whole class of the conditional constructor based rewrite systems. The strategy is implicitly defined by a transformation that allows the application of [4].

3 Conclusion

This paper recalls some fundamental rewriting and narrowing strategies. The presentation is new and unusual in that all the presented strategies are casted from the viewpoint of demandness. This viewpoint more easily allows to understand the conditions in which a strategy executes steps that are not executed by another strategy.

Our work is still in progress. Therefore this conclusion is more a prologue of future work than an epilogue of the achieved results. We plan to precisely formulate a notion, for defined operations, of an argument demanded by a rewrite rule. The canonical replacement maps of context sensitive, lazy and on-demand rewriting can all be expressed using this notion. To a large extent, more advanced strategies, such as demand driven, needed and weakly needed rewriting can expressed using this notion, as well.

These considerations suggest that the notion of an argument demanded by a rewrite rule is both atomic and fundamental to the study of strategies. By comparing how different strategies use this fundamental notion, we expect new insights on the behavior of a strategy and on the differences and similarities between related strategies.

References

1. M. Alpuente, M. Falaschi, P. Julian, and G. Vidal. Specialization of lazy functional logic programs. In *Proc. of PEPM'97*. ACM, 1997.
2. S. Antoy. Non-determinism and lazy evaluation in logic programming. In T. P. Clement and K.-K. Lau, editors, *Logic Programming Synthesis and Transformation (LOPSTR'91)*, pages 318–331, Manchester, UK, July 1991. Springer-Verlag.
3. S. Antoy. Definitional trees. In *Proc. of the 4th Intl. Conf. on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.
4. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of the 6th International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
5. S. Antoy. Constructor-based conditional narrowing. In *Proc. of the 3rd International Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 199–206, Florence, Italy, Sept. 2001. ACM.
6. S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. of the 14th International Conference on Logic Programming (ICLP'97)*, pages 138–152. MIT Press, 1997.
7. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, July 2000.
8. W. Fokkink, J. Kamperman, and P. Walters. Lazy rewriting on eager machinery. *ACM Transactions on Programming Languages and Systems*, 22(1):45–86, 2000.
9. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.

10. G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*, pages 395–443. MIT Press, Cambridge, MA, 1991.
11. R. Loogen, F. López Fraguas, and M. Rodríguez Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. 5th International Symposium on Programming Language Implementation and Logic Programming (PLILP'93)*, pages 184–200. Springer LNCS 714, 1993.
12. S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*. to appear.
13. S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1):191–223, January 1998.
14. S. Lucas. Termination of on-demand rewriting and termination of OBJ programs. In *Proc. of 3rd International Conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 82–93. ACM Press, 2001.
15. S. Lucas. Lazy rewriting and context-sensitive rewriting. In M. Hanus, editor, *Electronic Notes in Theoretical Computer Science*, volume 64. Elsevier Science Publishers, 2002. Available at `http://www.elsevier.nl/locate/entcs/volume64.html`.
16. J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.
17. M. J. O'Donnell. *Computing in Systems Described by Equations*. Springer LNCS 58, 1977.
18. S. Peyton Jones and J. Hughes (ed.). Haskell 98: A non-strict, purely functional language. Available at `http://www.haskell.org/onlinereport/`.
19. R. C. Sekar and I. V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. *Information and Computation*, 104(1):78–109, 1993.