Reduction Strategies for Declarative Programming

Michael Hanus

Institut für Informatik, Christian-Albrechts-Universität Kiel
D-24098 Kiel, Germany, `mh@informatik.uni-kiel.de`

Abstract. This paper surveys reduction or evaluation strategies for functional and functional logic programs. Reasonable reduction strategies for declarative languages must be efficiently implementable to be useful in practice. On the other hand, they should also support the programmers to write programs in a declarative way ignoring the influence of the evaluation strategy to the success of a computation as good as possible. We review existing reduction strategies along these lines and discuss some aspects for further investigation.

1 Background

Although term and graph rewriting is a universal framework to investigate operational principles for declarative (mainly functional) programming languages, research on particular reduction strategies is often done for general rewrite systems, i.e., the special properties required for programming languages are not exploited. For instance, the usual restriction on rules found in (functional as well as logic) programming languages is the requirement for *constructor-based rules*, i.e., each left-hand side must contribute to the definition of the function's semantics in a constructive way. Syntactically, this is ensured by allowing only constructor terms as arguments in each left-hand side. It has been shown in [6] that this requirement alone is sufficient to provide (rewrite, model-theoretic, fixpoint) semantics for a rather general declarative programming language. In particular, other conditions like confluence or termination are not required (and also not desirable from a programming language point of view, cf. [6]).

On the other hand, techniques in term rewriting are often motivated by problems in theorem proving or algebraic specifications where constructor-based rules are considered as too restricted. Therefore, the reduction strategies developed for general term rewriting systems had only a limited impact on the operational semantics of programming languages. For instance, most of the current functional languages are based either on a simple innermost strategy (eager languages like Standard ML) or a simple left-to-right lazy strategy (like in Haskell), although much more powerful reduction strategies are known. A consequence of the use of such simple strategies is a more operational rather than declarative view of programs. Since innermost rewriting is not normalizing, programmers of eager functional languages must carefully consider the influence of the innermost evaluation strategy to the success of a computation (e.g., introducing non-strict if-then-else expressions to avoid the evaluation of some subterms). Unfortunately, lazy strategies do not help very much when restricted to a simple left-to-right top-down pattern matching like in Haskell. For instance, if one writes in a Haskell program the equation "`f 1 = 1`", then the expression "`(f 1)`" might reduce to "2"![1] Furthermore, the standard reduction strategy might not terminate even for rewrite rules where a sequential normalizing reduction strategy is known, as in this example:

```
g 0 []     = 0
g x (y:ys) = y
```

Consider a non-terminating function $\perp$. Although the normal form of the expression `(g` $\perp$ `[1])` is `1` (by the second rule for `g`), Haskell does not terminate on this expression due to its strict left-to-right evaluation strategy which causes the non-terminating evaluation of $\perp$. As a consequence, rules in Haskell cannot be interpreted as equations but all the rules defining a function in a Haskell program must be passed through a complex pattern-matching compiler [17] in order to understand their meaning.

2 Improving Reduction Strategies to Support Declarative Programming

The general objective of declarative programming is to support the development of programs that are understandable without or with only a limited consideration of the program's execution. As shown above, the use of simple reduction strategies does not fully support this goal. Therefore, one has to find *reduction strategies that support a simple understanding as well as an efficient execution of declarative programs.* For the sake of a simple understanding, complex transformations to define the semantics of programs should be avoided. Ideally, each component of a program should contribute to the semantics of the entire program in a compositional manner. Thus, an equational reading of each program rule should be supported so that one can check the validity of the program by checking its individual parts. As a consequence, the textual order of program rules becomes less important. The exact kind of "equational reading" might depend on the particular programming language. For instance, a functional language based on confluent (e.g., orthogonal) rewrite systems could interpret program rules as equations in classical equational logic. However, a functional logic language allowing non-deterministic functions defined by non-confluent rewrite systems could be based on a rewrite logic where rewrite rules are only instantiated with constructor terms [6].

Apart from these details, a reasonable reduction strategy to evaluate expressions should be *normalizing*, i.e., it should compute a normal form w.r.t. the

[1] This strange behavior can happen if another equation like "`f x = 2`" occurs *before* the equation "`f 1 = 1`" in the program text.

rewrite logic whenever it exists (without a termination requirement on the set of rewrite rules). Such reduction strategies are known for a long time in term rewriting (e.g., [12]) but have not been considered in realistic programming languages (maybe due to the fact that they seem too complex in the general case of term rewriting). For the reasons discussed above, it is important to include more sophisticated reduction strategies in real programming languages. A normalizing strategy supports the simple understanding of programs. However, is it also possible to find an efficient strategy as well? Fortunately, a lot of pieces of research are now available to base new declarative programming languages on sophisticated efficient *and* normalizing reduction strategies, but some substantial further research is still required as discussed in the following.

One key idea to enable efficient evaluation strategies is the *restriction to constructor-based rules*. As discussed above, this is not a real restriction for programming languages but always satisfied if functions are defined in a constructive way. In the case of constructor-based rules, strongly sequential [12] and inductively sequential [1] rewrite systems are identical [10]. Strongly sequential rewrite systems are those (orthogonal) systems for which a relative efficient (i.e., sequential) reduction strategy exists. Since strongly sequential system are not necessarily constructor-based, the corresponding reduction strategy is more complicated than those used in current implementations of functional languages. However, for the interesting subclass of constructor-based rewrite systems, an efficient reduction strategy can be defined by a tree-like data structure, called *definitional trees* [1]. Reduction with definitional trees is *needed reduction*, i.e., only those redexes are evaluated that need to be evaluated in order to compute a normal form. Since definitional trees can be translated into standard case expressions [11], this reduction strategy can be implemented in any lazy functional language by replacing the standard left-to-right pattern matching of [17] by a more sophisticated pattern matcher [9]. For instance, the definition of function `g` in Section 1 can be translated into

```
g x1 x2 = case x2 of []     -> (case x1 of 0 -> 0)
                     (y:ys) -> y
```

so that the expression (`g` $\perp$ `[1]`) reduces to `1` in one step. The same strategy can also be used for *functional logic languages* where expressions might contain free variables during evaluation. Such variables are (non-deterministically) instantiated to constructor terms whenever this is necessary to proceed a computation (i.e., they occur as the first argument of a case expression). This evaluation strategy, called *needed narrowing* [4], is currently the best evaluation strategy for functional logic languages since it computes the shortest possible derivations and a minimal set of solutions (see [4] for details). Moreover, it reduces to a deterministic needed reduction strategy if free variables do not occur and can be efficiently implemented, e.g., by a translation into Prolog.

In inductively sequential systems, functions are inductively defined on the data structures they are working on. Although this is a very natural requirement so that most functions have inductively sequential definitions, in some applications (mainly applications written in a logic programming style) this is too restrictive. In a functional logic language, which already provides a mechanism for don't know non-determinism, there is one immediate extension of inductively sequential programs: allow several right-hand sides for one left-hand sides. This leads to the notion of *non-deterministic functions* which have the property that calls to such functions might have several normal forms. For instance, a call to the function `coin` defined by the rule

```
coin = 0 | 1
```

(where the vertical bar denotes an alternative between two expressions) reduces non-deterministically to the expression `0` or `1`. A declarative semantics for programs containing non-deterministic functions is defined in [6] and an operational semantics can be based on an extension of needed reduction with definitional trees [2].

Inductively sequential systems have the property that either a rule is applicable or there is a single argument position that must be reduced in order to compute a normal form. However, sometimes it is very natural (from a declarative point of view) to define functions in a form that does not satisfy this property. For instance, if we represent sets as lists, the intersection of two sets can be defined by the rules

```
intersection [] ys = []
intersection xs [] = []
intersection (x:xs) (y:ys) = ···
```

This natural definition does not contain a distinguished inductive argument position since a rule for `intersection` is applicable if the first *or* the second argument is reducible to `[]`. Thus, a sequential needed reduction strategy is not applicable and current programming languages treat such definitions as follows. Functional languages uses a kind of backtracking in pattern matching, i.e., initially the first argument is evaluated and, if this is not successful (i.e., not reducible to a demanded constructor), the second argument is evaluated. This has the drawback that a non-terminating evaluation of the first argument inhibits the evaluation of the entire call. (Functional) logic languages evaluates both arguments in independent disjunctions which can cause similar non-termination problems (when these disjunctions are implemented by backtracking as in Prolog) or superfluous computations. One solution to avoid these problems is the evaluation of both arguments in parallel, i.e., the extension of a sequential reduction strategy (which always select a single redex for the next reduction step) to a parallel reduction strategy which reduces in each step a *set* of redexes. For instance, in a function call of the form (`intersection` t_1 t_2), in each step both arguments t_1 and t_2 are stepwise reduced towards a normal form (if possible). Although such parallel reduction strategies have clear advantages and have been examined in [16] for functional programming and [3] for functional logic programming, traditional abstract machines for the efficient implementation of functional (logic) languages only support sequential strategies so that the efficient implementation of parallel reduction strategies needs some further investigation. Nevertheless, it is interesting to note that parallel reduction or narrowing strategies may lead to a more declarative programming style since the consequences of a particular formulation of the program rules w.r.t. a sequential strategy do not need to be considered. For instance, the textual ordering of rules is less important w.r.t. a parallel strategy in contrast to a sequential strategy.

In the context of functional logic languages, the reduction of the search space is similarly important as the requirement for a normalizing strategy in purely functional languages. However, the techniques to achieve this are still not sufficiently investigated. As shown in [3], there is a tradeoff between the size of the search space and the length of the derivations. For instance, one can reduce the number of different alternative substitutions computed for the next narrowing step before actually performing the alternative (parallel) narrowing steps. This reduces the breadth of the search tree but possibly increases the length of successful derivations. A technique to reduce the number of narrowing steps in a derivation is the inclusion of a *simplification phase between narrowing steps*: before a narrowing step is performed, the expression is reduced w.r.t. a set of simplification rules. This idea has been pioneered in the language SLOG [5] w.r.t. an innermost narrowing strategy and terminating rewrite rules but its adaption to a lazy evaluation strategy, where termination is not required, is less clear. One question is which kind of simplification strategy should be used. Considering the discussion above, a sophisticated reduction strategy (needed or parallel) is preferable. Another question is the selection of appropriate simplification rules. For the completeness of the narrowing strategy, it is important that the simplification phase is always terminating. This can be ensured by performing only a fixed finite number of simplification steps (e.g., one parallel reduction step) or a computation of normal forms w.r.t. a set of simplification rules with a terminating rewrite relation. Furthermore, one can also add *inductive consequences* to the set of simplification rules which can further reduce the search space (see the examples in [5] or [8]). Some of these options are discussed in [3], but the influence of the different techniques to practical applications needs further research. Furthermore, the efficient implementation of these techniques requires a deeper investigation.

Apart from the design and implementation of evaluation strategies for programs based on standard (constructor-based) rewrite rules, there are even more questions when one takes extensions of these rewrite rules into account that are desirable for application programming. These are (among others):

Higher-order features: From functional programming it is well known that higher-order functions improve code reuse and compositionality. Functional logic languages allow more possibilities to support higher-order features than purely functional languages due to the fact that a free variable can also denote a functional value. Moreover, it is also possible to allow lambda abstractions as patterns, e.g., to specify scoping rules in programming languages [11]. In this general case, higher-order unification can be used to instantiate free higher-order variables which is complete but computationally expensive. If one does not allow lambda abstractions as patterns, general higher-order unification is not needed but it is sufficient to instantiate free higher-order variables to all partial applications of functions defined in the program [7]. Although this reduces the complexity compared to higher-order unification, the guessing steps for free higher-order variables are still highly non-deterministic leading to huge search spaces. Another possibility is the delay of these guessing steps—an approach used in the multi-paradigm language Curry [9] (this requires an operational semantics supporting concurrent computations but has the risk of deadlocks, i.e., incomplete computations). Thus, there is a tradeoff between efficient handling, completeness and expressiveness of higher-order features in functional logic languages and the practical consequences of this tradeoff need some further research.

Default rules: The sequential top-to-bottom pattern matching strategy of functional languages has the advantage that default rules, i.e., rules that are applied when no other rule is applicable, can be easily defined by putting them textually after all other rules. Although such default rules are only an abbreviation for a set of rules with implicitly specified patterns and, therefore, they are conceptually not needed, they are very useful for application programming. Since in functional logic languages different rules lead to different computations and solutions, the semantics of default rules is less clear. Approaches to handle default rules are investigated, for instance, in [13, 14] The operational techniques to handle default rules in functional logic languages are much more involved than in purely functional languages so that their efficient treatment is less clear and needs further investigations.

Constraints: The advantages of using constraints is well known from many areas, mainly logic programming. In particular, they improve the "declarativeness" of programs by moving an explicit operational treatment of constraints into the constraint solver provided by the language's implementation. Thus, functional logic languages should also offer these advantages by including constraint structures into their computational domain. Although all these languages offer equational constraints, the inclusion of other constraint domains is less clear. In particular, the interaction of lazy evaluation with constraint solving is a new aspect w.r.t. purely (constraint) logic languages. [15] contains a recent proposal of functional logic programming with constraints.

3 Conclusions

We have discussed some known reduction strategies for functional and functional logic programs. Such declarative programs can be considered as constructor-based rewrite systems. The restriction to constructor-based rules enables the definition of efficient strategies for large classes of programs. We have also discussed how sophisticated reduction strategies can lead to a more declarative programming style since the programmer is less forced to consider the influence of the reduction strategy on the success of a computation. However, there are many topics for further research that need to be investigated before modern declarative languages can be fully based on such reduction strategies.

References

1. S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.
2. S. Antoy. Optimal Non-Deterministic Functional Logic Computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pp. 16–30. Springer LNCS 1298, 1997.
3. S. Antoy, R. Echahed, and M. Hanus. Parallel Evaluation Strategies for Functional Logic Languages. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pp. 138–152. MIT Press, 1997.
4. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000. Previous version in *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, 1994.
5. L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 172–184, Boston, 1985.
6. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, Vol. 40, pp. 47–87, 1999.
7. J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. A Higher Order Rewriting Logic for Functional Logic Programming. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pp. 153–167. MIT Press, 1997.
8. M. Hanus. Lazy Narrowing with Simplification. *Computer Languages*, Vol. 23, No. 2–4, pp. 61–85, 1997.
9. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
10. M. Hanus, S. Lucas, and A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, Vol. 67, No. 1, pp. 1–8, 1998.
11. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, Vol. 9, No. 1, pp. 33–75, 1999.
12. G. Huet and J.-J. Levy. Call by need computations in non-ambiguous linear term rewriting systems. Rapport de Recherche No. 359, INRIA, 1979.
13. F.J. López-Fraguas and J. Sánchez-Hernández. Proving Failure in Functional Logic Programs. In *Proc. First International Conference on Computational Logic (CL 2000)*, pp. 179–183. Springer LNAI 1861, 2000.
14. J.J. Moreno-Navarro. Default Rules: An Extension of Constructive Negation for Narrowing-based Languages. In *Proc. Eleventh International Conference on Logic Programming*, pp. 535–549. MIT Press, 1994.
15. M. Rodríguez-Artalejo. Functional and Cosntraint Logic Programming. In *CCL'99*, pp. 202–270. Springer LNCS 2002, 2001.
16. R.C. Sekar and I.V. Ramakrishnan. Programming in Equational Logic: Beyond Strong Sequentiality. *Information and Computation*, Vol. 104, No. 1, pp. 78–109, 1993.
17. P. Wadler. Efficient Compilation of Pattern-Matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pp. 78–103. Prentice Hall, 1987.