In *Proc. Sixth International Symposium on Programming Language Implementation and Logic Programming*, Madrid (Spain), pp. 370–384, Springer LNCS 844, 1994

Combining Lazy Narrowing and Simplification*

Michael Hanus

Max-Planck-Institut für Informatik
Im Stadtwald, D-66123 Saarbrücken, Germany.
`michael@mpi-sb.mpg.de`

Abstract. Languages that integrate functional and logic programming styles with a complete operational semantics are based on narrowing. In order to avoid useless computations and to deal with infinite data structures, lazy narrowing strategies have been proposed in the past. This paper presents an important improvement of lazy narrowing by incorporating deterministic simplification steps into lazy narrowing derivations. These simplification steps reduce the search space so that in some cases infinite search spaces are reduced to finite ones. We show that the completeness of lazy narrowing is not destroyed by the simplification process and demonstrate the improved operational behavior by means of several examples.

1 Introduction

In recent years, a lot of proposals have been made to amalgamate functional and logic programming languages [19]. Functional logic languages with a sound and complete operational semantics are based on *narrowing*, a combination of the reduction principle of functional languages and the resolution principle of logic languages. Narrowing, originally introduced in automated theorem proving [34], is used to *solve* equations by finding appropriate values for variables occurring in arguments of functions. This is done by unifying (rather than matching) an input term with the left-hand side of some rule and then replacing the instantiated input term by the instantiated right-hand side of the rule.

Example 1. Consider the following rules defining the addition of two natural numbers which are represented by terms built from `0` and `s`:

$$\begin{array}{rcll} \mathtt{0\ +\ N} & \to & \mathtt{N} & (R_1)\\ \mathtt{s(M)\ +\ N} & \to & \mathtt{s(M\ +\ N)} & (R_2)\end{array}$$

The equation `X+s(0)`$\approx$`s(s(0))` can be solved by a narrowing step with rule R_2 followed by a narrowing step with rule R_1 so that `X` is instantiated to `s(0)` and the instantiated equation is reduced to the trivial equation `s(s(0))`$\approx$`s(s(0))`:

$$\mathtt{X+s(0)}\approx\mathtt{s(s(0))} \leadsto_{\{\mathtt{X}\mapsto\mathtt{s(M)}\}} \mathtt{s(M+s(0))}\approx\mathtt{s(s(0))} \leadsto_{\{\mathtt{M}\mapsto\mathtt{0}\}} \mathtt{s(s(0))}\approx\mathtt{s(s(0))}$$

Hence we have found the solution `X`$\mapsto$`s(0)` to the given equation. □

* The research described in this paper was supported in part by the German Ministry for Research and Technology (BMFT) under grant ITS 9103 and by the ESPRIT Basic Research Working Group 6028 (Construction of Computational Logics). The responsibility for the contents of this publication lies with the author.

In order to ensure completeness in general, the left-hand side of *each* rule must be unified with *each* non-variable subterm of the given equation. Clearly, this yields a huge search space. The situation can be improved by particular narrowing strategies which restrict the possible positions for the application of the next narrowing step, e.g., basic [22], innermost [13], outermost [10], lazy [32], or needed narrowing [2]. In this paper we consider a *lazy narrowing* strategy where narrowing steps are applied at outermost positions in general and at an inner position only if it is demanded and contributes to some later narrowing step at an outer position. Similarly to pure functional programming, such a lazy strategy avoids some useless steps in comparison to an eager strategy. However, in the context of functional logic programming a lazy narrowing strategy can also have an unpleasant behavior if a demanded argument term has infinitely many head normal forms (i.e., if it can be derived to infinitely many terms with a variable or constructor at the top).

Example 2. Consider the following rules which may be part of a program for arithmetic operations:

`0 * N` $\rightarrow$ `0`	(R_3)	`one(0)` $\rightarrow$ `s(0)`	(R_5)
`N * 0` $\rightarrow$ `0`	(R_4)	`one(s(N))` $\rightarrow$ `one(N)`	(R_6)

If we want to compute a solution to the equation `one(X)*0`$\approx$`0` by lazy narrowing, we could try to apply rule R_3 to evaluate the left-hand side. In this case the first argument `one(X)` is demanded and must be evaluated to a term with a constructor at the top. Unfortunately, there are infinitely many possibilities to compute a head normal form `s(0)` of the term `one(X)` by instantiating `X` with $\underbrace{\mathtt{s(}\cdots\mathtt{s}}_{n}\mathtt{(0)}\cdots\mathtt{)}$ for arbitrary n. Hence lazy narrowing has an infinite search space in this example and does not compute a solution in a sequential implementation (see [15] for a discussion of problems with sequential implementations of lazy narrowing). However, we could avoid this infinite search space by computing the normal form of both sides of the equation before applying a narrowing step. The normal form of the initial equation is `0`$\approx$`0` (reduction of the left-hand side with rule R_4) which is trivially true. □

The idea of reduction to normal form before applying a narrowing step has been mainly proposed with respect to eager narrowing strategies [12, 13, 21, 30, 33]. It has been shown that eager narrowing with normalization is a more efficient control strategy than left-to-right SLD-resolution for equivalent logic programs [13, 18]. The main contribution of this paper is the combination of lazy narrowing with intermediate simplification steps. We show that this combination does not destroy the completeness of lazy narrowing and discuss its usefulness for various classes of functional logic programs. The previous example has shown that the integration of simplification can improve the operational behavior of lazy narrowing if there are rules with overlapping left-hand sides, but we will also provide examples where all rules have non-overlapping left-hand sides.

In the next section we recall basic notions from term rewriting and functional logic programming. In Section 3 we show how to include a deterministic simplification process into lazy narrowing derivations. Finally, we discuss in Section 4 the usefulness of this simplification process for different classes of functional logic programs.

2 Preliminaries

In this section we recall basic notions of term rewriting [7] and functional logic programming [19].

A *signature* is a set $\mathcal{F}$ of *function symbols*.[2] Every $f \in \mathcal{F}$ is associated with an *arity* n, denoted f/n. Let $\mathcal{X}$ be a countably infinite set of *variables*. Then the set $\mathcal{T}(\mathcal{F}, \mathcal{X})$ of *terms* built from $\mathcal{F}$ and $\mathcal{X}$ is the smallest set containing $\mathcal{X}$ such that $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ whenever $f \in \mathcal{F}$ has arity n and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. We write f instead of $f()$ whenever f has arity 0. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term t is called *ground* if $\mathcal{V}ar(t) = \emptyset$.

Usually, functional logic programs are *constructor-based*, i.e., a distinction is made between operation symbols to construct data terms, called *constructors*, and operation symbols to operate on data terms, called *defined functions* or *operations* (see, for instance, the functional logic languages ALF [16], BABEL [29], K-LEAF [14], SLOG [13]). Hence we assume that the signature $\mathcal{F}$ is partitioned into two sets $\mathcal{F} = \mathcal{C} \cup \mathcal{D}$ with $\mathcal{C} \cap \mathcal{D} = \emptyset$. A *constructor term* t is built from constructors and variables, i.e., $t \in \mathcal{T}(\mathcal{C}, \mathcal{X})$.

A (*rewrite*) *rule* $l \to r$ is a pair of terms l and r satisfying $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ where l has the form $f(t_1, \ldots, t_n)$ with $f \in \mathcal{D}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. l and r are called *left-hand side* and *right-hand side*, respectively.[3] A rule is called a *variant* of another rule if it is obtained by a unique replacement of variables by other variables. A *term rewriting system* $\mathcal{R}$ is a set of rules.

The execution of functional logic programs requires notions like substitution, unifier, position etc. A *substitution* σ is a mapping from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that the set $\{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is finite. We frequently identify a substitution σ with the set $\{x \mapsto \sigma(x) \mid \sigma(x) \neq x\}$. Substitutions are extended to morphisms on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ by $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$ for every term $f(t_1, \ldots, t_n)$. A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. A unifier σ is called *most general* (*mgu*) if for every other unifier σ' there is a substitution ϕ with $\sigma' = \phi \circ \sigma$ (concatenation of σ and ϕ). Most general unifiers are unique up to variable renaming. By introducing a total ordering on variables we can uniquely choose *the* most general unifier of two terms. A *position* p in a term t is represented by a sequence of natural numbers, $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s (see [7] for details).

[2] In this paper we consider only single-sorted programs. The extension to many-sorted signatures is straightforward [31]. Since sorts are not relevant to the subject of this paper, we omit them for the sake of simplicity.

[3] For the sake of simplicity we consider only unconditional rules, but our results can easily be extended to conditional rules with the restrictions of the functional logic language BABEL [29].

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{\mathcal{R}} s$ if there exist a position p in t, a rewrite rule $l \to r$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. In this case we say t is *reducible* (at position p). A term t is called *irreducible* or in *normal form* if there is no term s with $t \to_{\mathcal{R}} s$.

$\to^*_{\mathcal{R}}$ denotes the transitive-reflexive closure of the rewrite relation $\to_{\mathcal{R}}$. $\mathcal{R}$ is called *terminating* if there are no infinite rewrite sequences $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} t_3 \to_{\mathcal{R}} \cdots$. $\mathcal{R}$ is called *confluent* if for all terms t, t_1, t_2 with $t \to^*_{\mathcal{R}} t_1$ and $t \to^*_{\mathcal{R}} t_2$ there exists a term t_3 with $t_1 \to^*_{\mathcal{R}} t_3$ and $t_2 \to^*_{\mathcal{R}} t_3$.

If $\mathcal{R}$ is confluent and terminating, normal forms uniquely exist and we can decide the validity of an equation $s \approx t$ by computing the normal form of both sides using an arbitrary sequence of rewrite steps. In order to *solve* an equation, we have to find appropriate instantiations for the variables in s and t. This can be done by *narrowing*. A term t is *narrowable* into a term t' if there exist a non-variable position p in t (i.e., $t|_p \notin \mathcal{X}$), a variant $l \to r$ of a rewrite rule with $\mathcal{V}ar(t) \cap \mathcal{V}ar(l) = \emptyset$, a substitution σ such that σ is a most general unifier of $t|_p$ and l, and $t' = \sigma(t[r]_p)$. In this case we write $t \leadsto_\sigma t'$.[4]

Narrowing is able to solve equations w.r.t. $\mathcal{R}$ by deriving both sides of an equation to syntactically unifiable terms [22]. Due to the huge search space of simple narrowing, several authors have proposed restrictions on the admissible narrowing derivations like basic narrowing [22], innermost narrowing [13], or outermost narrowing [10]. *Lazy narrowing* [5, 27, 32] is influenced by the idea of lazy evaluation in functional programming languages. Lazy narrowing steps are applied to outermost positions with the exception that arguments are evaluated by narrowing to their head normal form if their values are demanded for an outermost narrowing step (see [29] for an exact definition of a lazy narrowing position). Lazy narrowing has at least two advantages in comparison to other narrowing strategies:

1. Since lazy narrowing applies narrowing steps at inner positions only if it is demanded by some rule, useless narrowing steps (steps at inner positions which do not contribute to the result) are avoided.[5]
2. Since lazy narrowing evaluates functions only if their results are demanded, it can deal with nonterminating functions and infinite data structures. The other narrowing strategies cited above require a terminating set of rewrite rules and cannot deal with infinite data structures.

The next example should emphasize the latter point.

Example 3. The following rules define a function `from(N)` which computes an infinite list of naturals starting from `N` and a function `first(N,L)` which computes the first `N` elements of the list `L`:

```
from(N) → [N|from(s(N))]
```

[4] Since the instantiation of the variables in the rule $l \to r$ by σ is not relevant for the computed solution of a narrowing derivation, we will omit this part of σ in the example derivations in this paper.

[5] To be precise, the avoidance of useless narrowing steps depends on the lazy narrowing strategy. Although this is one of the motivations of all lazy strategies, the only strategy for which this property has been formally proved is needed narrowing [2].

```
         first(0,L) → []
    first(s(N),[E|L]) → [E|first(N,L)]
```

Then lazy evaluation of the expression `first(s(s(0)),from(0))` yields the result `[0,s(0)]` while an eager evaluation does not terminate due to the nonterminating eager evaluation of `from(0)`. Similarly, lazy narrowing applied to the equation `first(X,from(Y))`$\approx$`[0,s(0)]` computes the solution $\{$`X`$\mapsto$`s(s(0))`,`Y`$\mapsto$`0`$\}$ while an eager narrowing strategy runs into an infinite loop. □

Since narrowing applies rules only in one direction from left to right, the confluence of the rewrite relation is an essential requirement for the completeness of all narrowing strategies. But confluence is an undecidable property of a rewrite system if it is not terminating. Therefore functional logic languages based on a lazy evaluation strategy have the following requirements on the rewrite rules in order to ensure completeness:

1. *Left-linearity:* All rules are left-linear, i.e., no variable appears more than once in the left-hand side of any rule.
2. *Nonambiguity:* If $l_1 \to r_1$ and $l_2 \to r_2$ are two different rules, then l_1 and l_2 are not unifiable (*strong nonambiguity*), or if l_1 and l_2 have a most general unifier σ, then $\sigma(r_1)$ and $\sigma(r_2)$ are identical (*weak nonambiguity*).

These conditions ensure the uniqueness of normal forms if they exist. Due to the presence of nonterminating functions, the completeness results for lazy strategies are stated with respect to domain-based interpretations of rewrite rules [14, 29]. In particular, the equality of two expressions holds only if both sides are reducible to the same ground constructor term.

The nonambiguity condition does not exclude applications from logic programming. In fact, if we allow also conditional rules (as in BABEL [29]), any logic program can be translated into a set of weakly nonambiguous rules by representing predicates as Boolean functions [29].

Another important improvement of simple narrowing is *normalizing narrowing* [12] where the term is rewritten to its normal form before a narrowing step is applied. This optimization is important since it prefers deterministic computations: rewriting a term to normal form can be done in a deterministic way since every rewriting sequence yields the same result (if $\mathcal{R}$ is confluent and terminating). As shown in [13, 18], normalizing narrowing has the effect that functional logic programs are more efficiently executable than pure logic programs. It has been shown that normalization can also be combined with other eager narrowing strategies. Réty [33] has proved completeness of *normalizing basic narrowing*, Fribourg [13] has proposed *normalizing innermost narrowing* and Hölldobler [21] has combined innermost basic narrowing with normalization. Because of these advantages, normalizing narrowing is the foundation of several programming languages which combines functional and logic programming like ALF [16], LPG [3] or SLOG [13]. However, normalization has not been included in lazy narrowing strategies. Therefore we will show that deterministic simplification steps could be performed before nondeterministic lazy narrowing steps without destroying the completeness of lazy narrowing. The problems of integrating normalization into basic narrowing [33] shows that such a result is not obvious.

3 Integrating Simplification into Lazy Narrowing

In this section we show that deterministic simplification steps can be included in lazy narrowing derivations without destroying completeness. Since we are interested in a lazy narrowing strategy, we consider a functional logic program consisting of a constructor-based term rewriting system $\mathcal{R}$ which satisfies the left-linearity and nonambiguity condition.

Loogen and Winkler [26] have shown how to increase deterministic computations in the implementation of such programs: if no goal variable has been bound in a narrowing step, then all attempts to apply alternative rules at the same position can be ignored due to the nonambiguity of the rules. In this case a "cut" can be executed to eliminate the choice point for alternative rules. Since the execution of this "cut" depends on the run-time behavior of the program (whether or not a goal variable has been bound during unification), it is called *dynamic cut* in [26]. The dynamic cut can be implemented by a special `POP` instruction which checks whether a goal variable has been bound during unification and, if this did not happen, removes the last choice point. The advantage of this method is its simple implementation, but it has also two disadvantages:

1. The dynamic cut removes choice points which have been created but are not needed in the further computation process. Hence it does not avoid the creation of choice points (one of the most expensive operations in the implementation of logic languages): if a choice point is not needed in a deterministic computation, it is created and then deleted after the unification of the rule's left-hand side.
2. The detection of deterministic computations depends on the order of the rules. If a rule which enables a deterministic computation step is not at the beginning, nondeterministic steps may be performed even if a deterministic step is possible.

The second disadvantage is discussed in more detail in the following example.

Example 4. Consider the rules of Example 2 and the goal equation `0*one(X)`≈`0`. Using the dynamic cut technique, first a choice point for the rules R_3 and R_4 is created, then rule R_3 is applied to narrow the left-hand side yielding the trivial equation `0`≈`0`, and after that the choice point is removed since no goal variable (`X`) has been bound in the narrowing step (dynamic cut). Hence the attempt to apply rule R_4 is avoided by the dynamic cut. But if we try to solve the equation `one(X)*0`≈`0`, the dynamic cut has no effect. As before, first a choice point for the rules R_3 and R_4 is created, then an attempt to apply rule R_3 is made.[6] Since it is necessary to evaluate the first argument in order to decide the applicability of this rule, `one(X)` is a lazy narrowing redex which is evaluated by applying rules R_5 or R_6 (this evaluation has an infinite search space and does not terminate in a sequential implementation, cf. Example 2). In any case the goal variable `X` will be bound and therefore the dynamic cut has no effect. □

[6] Note that we consider a sequential implementation where the rules are applied in the given textual order.

Although the dynamic cut has some disadvantages since it is applied *after* a narrowing attempt, the nonambiguity of the rules is the key to exploit deterministic computations in functional logic programs. In the following we will show that we can apply deterministic rewrite steps before a narrowing step. This technique avoids the creation of superfluous choice points and is independent on the order of rules (if we use all rules also for rewrite steps).

The next lemma is due to Loogen and Winkler [26] and shows that it is not necessary to consider alternative rules for narrowing if one rule is applicable without binding goal variables. This is a consequence of the nonambiguity condition on rewrite rules.

Lemma 1. *Let $R_1 = l_1 \to r_1$ and $R_2 = l_2 \to r_2$ be two different program rules and t be a term which has no variables in common with R_1 and R_2. If $\sigma(l_1) = t$, i.e., t is narrowable by rule R_1 without instantiating any goal variables, then rule R_2 does not need to be considered, because either R_2 is not applicable or the result of applying R_2 yields an instance of the application of R_1.*

Hence we could try to match the left-hand side of some rule with the current goal before applying a narrowing step. If this is possible, we can perform the corresponding rewrite step and, by the previous lemma, ignore all other rules, i.e., we perform a deterministic computation step. Although this solves the problems exemplified in Example 4, it is not sufficient to exploit many possible deterministic computations. In general, rewrite steps must also be performed at inner positions in order to enable rewrite steps at outer positions. For instance, consider the rules of Examples 1 and 2 and the goal equation `(0+0)*N≈0`. A rewrite step by applying rules R_3 or R_4 to the left-hand side of the equation is not possible. Hence we try to perform a narrowing step, i.e., generate a choice point for the rules R_3 or R_4, and so on. However, if we apply a rewrite step to the subterm `(0+0)` before the narrowing attempt, the equation is simplified to `0*N≈0` using rule R_1, and we could further simplify the equation to the trivial one `0≈0` using rule R_3. Therefore we could solve the equation without any nondeterministic narrowing step. The following lemma justifies deterministic rewrite steps at inner positions.

Lemma 2. *Let t, t' be terms such that $t \to_{\mathcal{R}} t'$ is a rewrite step at position p.*

1. *It is not necessary to consider alternative rules applied to t at position p.*
2. *All narrowing rules which are applicable to t at a position p', where $p' \neq p$ is a position not below p, are also applicable to t'.*

The applicability of narrowing rules at positions below p does not need to be considered: Due to the lazy narrowing strategy, narrowing steps at such positions would only be performed in order to apply some step at position p, but Proposition 1 of this lemma states that this is unnecessary since alternative rules do not need to be considered at position p.

Proof. Proposition 1 follows from Lemma 1 applied to position p. Proposition 2 is a consequence of the requirement for constructor-based rules: the subterm $t|_p$ must have a defined function symbol at the top since $t \to_{\mathcal{R}} t'$ is a rewrite step at position p. If a narrowing rule is applicable to t at position p', i.e., there is a rule $l \to r$ and a mgu σ of $t|_{p'}$ and l, and p' is a position above p (the case of independent positions is trivial since variables in t are not instantiated by the rewrite step), then there must be a variable position p'' in l (i.e., $l|_{p''} \in \mathcal{X}$) such that $\sigma(l)|_{p''}$ contains the subterm $t|_p$ (since all proper subterms of l contain only constructors and variables). But then there is also a unifier σ' of $t'|_{p'}$ and l which can be obtained by modifying σ for the variable $l|_{p''}$ (note that l has no multiple occurrences of variables). Hence we can apply rule $l \to r$ to t' at position p'. □

As a consequence of this lemma we can deterministically apply rewrite rules at any position before a narrowing step. The simplification of the goal by rewrite rules can be done in any order and in any depth. For instance, if the set of rewrite rules is terminating, normal forms uniquely exist and can be computed by repeated application of rewrite steps in any order until no more rewrite steps are applicable. This approach has been taken in normalizing narrowing [12, 13, 21, 30, 33] and in the functional logic languages ALF [16], LPG [3] and SLOG [13]. However, in the presence of nonterminating functions, an arbitrary simplification process could destroy the completeness of lazy narrowing as the following example shows.

Example 5. Consider the rules of Example 3 and the term `first(s(0),from(0))`. Lazy narrowing reduces this goal term to the term `[0]`. If we allow arbitrary simplification steps, we could apply infinitely many rewrite steps to evaluate the subterm `from(0)`:

$$
\begin{aligned}
\texttt{first(s(0),from(0))} &\to_{\mathcal{R}} \texttt{first(s(0),[0|from(s(0))])} \\
&\to_{\mathcal{R}} \texttt{first(s(0),[0,s(0)|from(s(s(0)))])} \\
&\to_{\mathcal{R}} \cdots
\end{aligned}
$$

Hence we would run into an infinite loop instead of computing the normal form of the initial term. □

In order to avoid such problems and to do not introduce additional superfluous work by the simplification process, we require to perform simplification steps lazily with the same strategy as narrowing, i.e., we consider the combination of *lazy narrowing with lazy simplification*. Since rewrite steps are also particular narrowing steps, an infinite loop caused by simplification occurs in lazy narrowing derivations without simplification, too. The only difference is that the order of rule applications in simplification steps may be different from the order of rule applications in narrowing steps. Hence it may be the case that the simplification process runs into an infinite loop while lazy narrowing without simplification first computes an answer and then runs into an infinite loop.

Example 6. Consider the rules of Example 2 and the following rule defining a nonterminating function:

`f(0)` $\to$ `f(0)`

If the goal equation `X*f(0)`$\approx$`0` should be solved, a lazy simplification strategy tries to evaluate the subterm `f(0)` to the constructor `0` in order to apply rule

R_4 to the left-hand side of the equation. Since the evaluation of `f(0)` loops, the simplification process does not terminate and no solution is computed. On the other hand, lazy narrowing without simplification narrows the left-hand side of the equation by applying rule R_3. This binds goal variable `X` to `0` and yields the trivial equation `0`≈`0`. However, after the computation of this solution an attempt to apply the alternative rule R_4 to the left-hand side is made which yields the same infinite loop as in the simplification process. □

Note that this different behavior is due to a particular sequential implementation of the strategy. In an implementation which collects all answers until the entire search space has been examined we would obtain no answer in both cases due to the infinite search space.

A simple solution to avoid a nonterminating simplification process is the inclusion of a *terminating subset* of the program rules for simplification. Since lazy narrowing is already complete without simplification, it is not necessary to perform rewrite steps with all possible program rules but we can arbitrarily restrict the set of rules used for rewrite steps. In the light of the previous example it is a reasonable decision to include a rule set with a terminating rewrite relation for simplification. This ensures the termination of the simplification process. The selection of this subset of rewrite rules could be done by the programmer or by the system (e.g., include only those rewrite rules for which a termination proof can be constructed). We have made the experience that for most practical examples termination proofs can be automatically constructed using syntactic termination orderings from term rewriting [6]. This is the case for all rules presented so far (of course, except for the `first`-rule of Example 3 and the `f`-rule of Example 6). An example where a terminating subset of all program rules is used for simplification will be given in Section 4.3.

4 Application to Functional Logic Programs

In this section we discuss the usefulness of integrating simplification into lazy narrowing derivations with respect to different classes of functional logic programs. In general, we consider constructor-based rewrite systems satisfying the left-linearity and nonambiguity conditions. However, there are important subclasses of such rewrite systems with different implications on the usefulness of integrating simplification. In this section we consider the following three subclasses in more detail: inductively sequential systems [1] where the rules for each function can be organized in a hierarchical structure, orthogonal systems satisfying the strong nonambiguity condition (no overlapping in the left-hand sides of the rules), and weakly orthogonal systems with overlapping left-hand sides.

4.1 Inductively Sequential Programs

In many functional as well as functional logic programs functions are defined by a case distinction on the different constructors occurring in the data type of the arguments. For instance, the definition of the addition function on natural numbers (cf. Example 1) is based on a case distinction for the first argument with respect to the constructors `0` and `s`. As another example consider the following rules defining a less-or-equal function on naturals:

$$
\begin{array}{rcll}
\texttt{0} & \leq \texttt{X} & \rightarrow \texttt{true} & (R_1) \\
\texttt{s(X)} & \leq \texttt{0} & \rightarrow \texttt{false} & (R_2) \\
\texttt{s(X)} & \leq \texttt{s(Y)} & \rightarrow \texttt{X} \leq \texttt{Y} & (R_3)
\end{array}
$$

Here is the main case distinction on the constructors of the first argument: if this argument is `0`, then only rule R_1 is applicable. If this argument has the constructor `s` at the top, then a further case distinction on the second argument is necessary to distinguish between rules R_2 and R_3. Altogether, the rules can be organized in a hierarchical structure representing the various case distinctions. Such hierarchical structures have been introduced by Antoy [1] under the name *definitional trees*. A program for which the rules of each function symbol can be organized in a definitional tree is called *inductively sequential*. Antoy, Echahed and Hanus [2] have defined for inductively sequential programs a narrowing strategy, called *needed narrowing*, which is optimal in the following sense: (1) it reduces only needed subterms in a narrowing step, i.e., subterms which must be reduced in any possible successful narrowing derivation, (2) it computes the shortest narrowing derivations if common subterms are shared, and (3) the solutions computed by two different narrowing derivations are independent. The needed narrowing steps are computed using the structure of definitional trees. Thus it can be efficiently implemented by pattern matching, and the strategy has an outermost (lazy) behavior.

Due to the optimality of needed narrowing the natural question arises whether the inclusion of simplification has an effect for this class of programs. To answer this question we recall the applicability conditions for a rewrite step. A functional expression can be reduced by a rewrite step if the arguments of the function call are sufficiently instantiated such that the left-hand side of some rule can be matched with the current call. Since the program is inductively sequential, there is always at most one rule matching the current call and this rule will be selected in the next narrowing step without instantiating any goal variables (see [2] for a detailed description of the strategy). Therefore a possible lazy reduction step is also computed by the needed narrowing strategy as a narrowing step, i.e., the inclusion of simplification steps has no effect. This is formally justified by the following proposition.

Proposition 3. *Let $\mathcal{R}$ be a set of inductively sequential rules. Then the integration of simplification does not shorten any needed narrowing derivation.*

Proof. By definition, rewrite steps are also particular narrowing steps. Thus any narrowing derivation with intermediate simplification steps is also a pure narrowing derivation. Since needed narrowing computes the shortest narrowing derivations [2], simplification cannot shorten any needed narrowing derivation. □

Hence it is unnecessary to integrate simplification in narrowing derivations for the class of inductively sequential programs.

4.2 Orthogonal Programs

The main example where we have demonstrated the improvements of simplification with respect to lazy narrowing (Example 2) has the property that two rules have overlapping left-hand sides. In the following we will show that the inclusion of simplification is useful even if there are no overlapping rules.

Example 7. Consider the following rewrite rules:

`f(0,s(M),N)` $\to$ `0`	(R_1)	`one(0)` $\to$ `s(0)`	(R_4)
`f(s(M),N,0)` $\to$ `0`	(R_2)	`one(s(N))` $\to$ `one(N)`	(R_5)
`f(N,0,s(M))` $\to$ `0`	(R_3)		

This is a orthogonal term rewriting system since all rules are left-linear and do not overlap in the left-hand sides. However, it is not inductively sequential since there is no argument which represents a case distinction on the constructors `0` and `s`. In fact, simplification has an important effect if we consider the goal equation `f(one(X),0,s(0))`≈`0`. Naive lazy narrowing first tries to apply rule R_1 to the left-hand side of this equation. Since the first argument of the rule's left-hand side is `0`, the evaluation of the actual argument `one(X)` is required in order to decide the unifiability of the first argument.[7] Similarly to Example 2, the evaluation of `one(X)` has an infinite search space and a sequential implementation does not compute any result since all evaluations of `one(X)` yields `s(0)` as the result which is not unifiable with the demanded value `0`. But if we simplify the goal equation before the attempt to apply a narrowing step, we use rule R_3 for a rewrite step which yields the trivial equation `0`≈`0`. Hence the infinite search space is avoided. □

4.3 Weakly Orthogonal Programs

In Sections 4.1 and 4.2 we have shown that the boundary of the usefulness of simplification in lazy narrowing derivations is between inductively sequential and orthogonal systems. We conjecture that for practical applications the most interesting class where simplification is useful is the class of weakly orthogonal programs which have rules with overlapping left-hand sides. Example 2 contains such a simple program, but the recursively defined constant function `one` may not convince the reader. Therefore we will demonstrate the positive effects of simplification by a more natural example.

Example 8. [20] Consider the following rules defining the Boolean operator ∨ and the predicate `even` on natural numbers:

`true` ∨ `B` $\to$ `true`	(R_1)	`even(0)` $\to$ `true`	(R_4)
`B` ∨ `true` $\to$ `true`	(R_2)	`even(s(0))` $\to$ `false`	(R_5)
`false` ∨ `false` $\to$ `false`	(R_3)	`even(s(s(X)))` $\to$ `even(X)`	(R_6)

This rewrite system is weakly orthogonal since rules R_1 and R_2 overlap. Now consider the goal equation `even(Z)`∨`true`≈`true` (note that this goal equation

[7] We assume that arguments are unified from left to right, otherwise a similar example can be constructed.

could also be the result of the more general equation $\texttt{even(Z)}\vee\texttt{B}\approx\texttt{true}$ where the Boolean variable B has been bound to true in the preceding computation). Naive lazy narrowing without simplification tries to apply a narrowing step with rule R_1. Since the value of the first $\vee$-argument is demanded by this rule, the subterm even(Z) is evaluated to a constructor-headed term by narrowing. There are infinitely many possibilities to do this, in particular the constructor true is derived by instantiating variable Z with the values $\texttt{s}^{2*i}\texttt{(0)}$, $i \geq 0$. Therefore lazy narrowing without simplification has an infinite search space and computes the additional special solutions $\{\texttt{Z}\mapsto\texttt{s}^{2*i}\texttt{(0)}\}$. On the other hand, if the equation is first simplified by applying rule R_2 to the left-hand side, we immediately obtain the trivial equation $\texttt{true}\approx\texttt{true}$ and avoid the infinite search space. □

We have mentioned that our method is complete even in the presence of nonterminating functions if a terminating subset of the program rules is used for simplification. This is demonstrated by a modification of the previous example.

Example 9. Consider the rules for $\vee$ of Example 8 (R_1, R_2, R_3) and the following new rules for not, even and odd:

not(true)	$\rightarrow$	false	(R_4)	even(X)	$\rightarrow$	not(odd(X))	(R_6)
not(false)	$\rightarrow$	true	(R_5)	odd(X)	$\rightarrow$	not(even(X))	(R_7)

Although even and odd are nonterminating functions, it is an admissible program. We use the terminating subset of the rules $\{R_1, R_2, R_3, R_4, R_5\}$ for simplification.[8] Consider the goal equation $\texttt{even(Z)}\vee\texttt{not(false)}\approx\texttt{true}$. Lazy narrowing without simplification tries to compute the head normal form of the subterm even(Z) since its value is demanded by rule R_1. Since this computation is nonterminating, naive lazy narrowing has an infinite search space. The same holds for lazy narrowing with the dynamic cut operator [26]. But lazy narrowing with simplification tries to apply rewrite steps first. No simplification rule is applicable to the entire left-hand side of the goal equation since the arguments are not in head normal form. Due to the lazy simplification strategy, we try to evaluate the arguments by simplification steps. The subterm even(Z) cannot be further simplified since rule R_6 is not included in the set of simplification rules. The second argument not(false) can be simplified to true by R_5 which causes the simplification of the complete left-hand side to true by R_2. Hence we obtain the trivial equation $\texttt{true}\approx\texttt{true}$. Thus the infinite search space is avoided. □

5 Conclusions and Related Work

In this paper we have shown how to improve the execution mechanism of functional logic programs. The basic idea is the integration of a deterministic simplification process into lazy narrowing derivations. This can be done in a simple way by using the program rules or a terminating subset of the program rules as simplification rules. The simplification strategy should be identical to the narrowing strategy. For particular and practically important classes of functional logic

[8] Note that the termination property of this subset can be automatically checked.

programs (orthogonal and weakly orthogonal programs) this has the positive effect that the search space is reduced without destroying completeness. Although we have emphasized the effect of simplification to the search space, the inclusion of simplification can also have an effect on the run time even if the search space is not reduced. For instance, if all program rules are used for simplification, ground goals are evaluated by simplification without generating any choice point while a lazy narrowing implementation would generate (and afterwards delete) choice points. Hence lazy narrowing with simplification combines the features from functional and logic programming also from an implementational point of view.

We have mentioned in the introduction and in Section 2 that the idea of exploiting deterministic computations by including simplification in functional logic languages has been proposed mainly for eager narrowing strategies like basic [30, 33], innermost [13] or innermost basic narrowing [21]. Echahed [11] has shown how to integrate normalization (with inductive consequences) in any narrowing strategy, but he requires strong restrictions on the set of rules (termination and uniformity, which is stronger than inductive sequentiality). The inclusion of simplification into lazy strategies has been considered only in the context of lazy unification.[9] In [8, 20] lazy unification calculi are proposed where terms are reduced to their normal form before a nondeterministic transformation step is applied to the equation system. But these approaches require a terminating set of rules in order to ensure the existence of normal forms and the completeness of the calculi.

As far as we know, the present paper is the first attempt to include simplification into narrowing derivations even in the presence of nonterminating functions. The only related work for this class of programs is the paper of Loogen and Winkler [26] which proposes the dynamic cut to detect deterministic narrowing steps after the unification phase. As discussed at the beginning of Section 3, this does not avoid the generation of choice points, and the cut of infinite derivation paths depends on the order of rules. The basic difference of our method is that we check the applicability of a deterministic computation step before we apply a nondeterministic step. Hence we prefer deterministic computations to nondeterministic computations. This qualifies our execution method as the operational principle of efficient functional logic languages.

Loogen et al. [25] have proposed to improve lazy narrowing strategies by reordering the unification steps in rule applications. For this purpose they use a version of definitional trees [1] extended to weakly orthogonal rewrite systems. In order to handle partial overlapping left-hand sides, they introduce nondeterministic choice nodes in definitional trees. But these choice nodes have the effect that possible deterministic computations are not detected. For instance, the infinite

[9] The combination of lazy narrowing with deterministic reduction steps has been also considered by Josephson and Dershowitz [23]. However, they provide no completeness proof but refer to [9] where only the completeness of naive narrowing without simplification and without a particular lazy strategy is proved for terminating conditional rules.

search spaces of naive lazy narrowing in Examples 2, 7 and 8 would also occur with respect to their improved strategy.

Another alternative to improve lazy narrowing has been proposed by Moreno-Navarro et al. [28]. They use information about demanded arguments to avoid reevaluations of expressions during unification with different rules. Since they do not change the order of argument evaluations and rules, the infinite search spaces avoided by simplification still occur in their approach.

The integration of simplification into lazy narrowing derivations requires new implementation techniques for functional logic languages. Current efficient implementations of lazy narrowing are mainly based on extensions of reduction machines used for the implementation of functional languages [4, 15, 24, 27]. The inclusion of simplification requires the implementation of an intermediate reduction process. This could be done by techniques proposed for the efficient implementation of normalizing narrowing [16, 17] or by the implementation of demons waiting for the sufficient instantiation of function arguments [23].

References

1. S. Antoy. Definitional Trees. In *Proc. of the 3rd Int. Conf. on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.
2. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, pp. 268–279, Portland, 1994.
3. D. Bert and R. Echahed. Design and Implementation of a Generic, Logic and Functional Programming Language. In *Proc. European Symposium on Programming*, pp. 119–132. Springer LNCS 213, 1986.
4. M.M.T. Chakravarty and H.C.R. Lock. The Implementation of Lazy Narrowing. In *Proc. of the 3rd Int. Symp. on Programming Language Implementation and Logic Programming*, pp. 123–134. Springer LNCS 528, 1991.
5. J. Darlington and Y. Guo. Narrowing and unification in functional programming - an evaluation mechanism for absolute set abstraction. In *Proc. of the Conf. on Rewriting Techniques and Applications*, pp. 92–108. Springer LNCS 355, 1989.
6. N. Dershowitz. Termination of Rewriting. *J. Symbolic Computation*, Vol. 3, pp. 69–116, 1987.
7. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pp. 243–320. Elsevier, 1990.
8. N. Dershowitz, S. Mitra, and G. Sivakumar. Equation Solving in Conditional AC-Theories. In *Proc. of the 2nd Int. Conf. on Algebraic and Logic Programming*, pp. 283–297. Springer LNCS 463, 1990.
9. N. Dershowitz and D.A. Plaisted. Equational Programming. In *Machine Intelligence 11*, pp. 21–56. Oxford Press, 1988.
10. R. Echahed. On Completeness of Narrowing Strategies. In *Proc. CAAP'88*, pp. 89–101. Springer LNCS 299, 1988.
11. R. Echahed. Uniform Narrowing Strategies. In *Proc. of the 3rd Int. Conf. on Algebraic and Logic Programming*, pp. 259–275. Springer LNCS 632, 1992.
12. M.J. Fay. First-Order Unification in an Equational Theory. In *Proc. 4th Workshop on Automated Deduction*, pp. 161–167, Austin (Texas), 1979. Academic Press.
13. L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Int. Symp. on Logic Programming*, pp. 172–184, Boston, 1985.
14. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A Logic plus Functional Language. *Journal of Computer and System Sciences*, Vol. 42, No. 2, pp. 139–185, 1991.

15. W. Hans, R. Loogen, and S. Winkler. On the Interaction of Lazy Evaluation and Backtracking. In *Proc. of the 4th Int. Symp. on Programming Language Implementation and Logic Programming*, pp. 355–369. Springer LNCS 631, 1992.
16. M. Hanus. Compiling Logic Programs with Equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 387–401. Springer LNCS 456, 1990.
17. M. Hanus. Efficient Implementation of Narrowing and Rewriting. In *Proc. Int. Workshop on Processing Declarative Knowledge*, pp. 344–365. Springer LNAI 567, 1991.
18. M. Hanus. Improving Control of Logic Programs by Using Functional Logic Languages. In *Proc. of the 4th Int. Symp. on Programming Language Implementation and Logic Programming*, pp. 1–23. Springer LNCS 631, 1992.
19. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *To appear in Journal of Logic Programming*, 1994. Also available as Technical Report MPI-I-94-201, Max-Planck-Institut für Informatik, Saarbrücken.
20. M. Hanus. Lazy Unification with Simplification. In *Proc. 5th European Symposium on Programming*, pp. 272–286. Springer LNCS 788, 1994.
21. S. Hölldobler. *Foundations of Equational Logic Programming.* Springer LNCS 353, 1989.
22. J.-M. Hullot. Canonical Forms and Unification. In *Proc. 5th Conf. on Automated Deduction*, pp. 318–334. Springer LNCS 87, 1980.
23. A. Josephson and N. Dershowitz. An Implementation of Narrowing. *Journal of Logic Programming (6)*, pp. 57–77, 1989.
24. R. Loogen. Relating the Implementation Techniques of Functional and Functional Logic Languages. *New Generation Computing*, Vol. 11, pp. 179–215, 1993.
25. R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of the 5th Int. Symp. on Programming Language Implementation and Logic Programming*, pp. 184–200. Springer LNCS 714, 1993.
26. R. Loogen and S. Winkler. Dynamic Detection of Determinism in Functional Logic Languages. In *Proc. of the 3rd Int. Symp. on Programming Language Implementation and Logic Programming*, pp. 335–346. Springer LNCS 528, 1991.
27. J.J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy Narrowing in a Graph Machine. In *Proc. Second Int. Conf. on Algebraic and Logic Programming*, pp. 298–317. Springer LNCS 463, 1990.
28. J.J. Moreno-Navarro, H. Kuchen, J. Marino-Carballo, S. Winkler, and W. Hans. Efficient Lazy Narrowing Using Demandedness Analysis. In *Proc. of the 5th Int. Symp. on Programming Language Implementation and Logic Programming*, pp. 167–183. Springer LNCS 714, 1993.
29. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, Vol. 12, pp. 191–223, 1992.
30. W. Nutt, P. Réty, and G. Smolka. Basic Narrowing Revisited. *Journal of Symbolic Computation*, Vol. 7, pp. 295–317, 1989.
31. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.
32. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Int. Symp. on Logic Programming*, pp. 138–151, Boston, 1985.
33. P. Réty. Improving basic narrowing techniques. In *Proc. of the Conf. on Rewriting Techniques and Applications*, pp. 228–241. Springer LNCS 256, 1987.
34. J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. *Journal of the ACM*, Vol. 21, No. 4, pp. 622–642, 1974.