In *Proc. 3rd International Conference on Algebraic and Logic Programming*,
pp. 228–243, Springer LNCS 632, 1992

Incremental Rewriting in Narrowing Derivations

Michael Hanus

Max-Planck-Institut für Informatik
Im Stadtwald
W-6600 Saarbrücken, Germany
e-mail: michael@mpi-sb.mpg.de

Abstract. The operational semantics of many proposals for the integration of functional and logic programming languages is based on narrowing. In order to reduce the search space and to prefer deterministic computations, the goal is rewritten to normal form between narrowing steps (normalizing narrowing). This rewriting process may be costly since the entire goal must be reduced to normal form after each narrowing step. We propose a useful optimization of the rewriting process: since the goal is in normal form before the narrowing step is applied and the narrowing step changes only small parts of the goal, rewriting can be restricted to a small number of positions in the narrowed goal in order to compute a new normal form. This optimization can speed up the execution mechanism of programming languages based on normalizing narrowing like SLOG or ALF.

1 Introduction

During recent years a lot of proposals have been made to amalgamate functional and logic programming languages [DL86, BL86]. A sound and complete operational semantics of such integrated languages is based on narrowing, a combination of the reduction principle of functional languages and the resolution principle of logic languages. Narrowing was originally introduced in automated theorem proving [Sla74]. In our context narrowing 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. For instance, consider the following rules defining the addition of two natural numbers which are represented by terms built from `0` and `s`:

```
0 + N    → N
s(M) + N → s(M + N)
```

We can solve the equation `X+s(0) = s(s(0))` by a narrowing step with the second rule followed by a narrowing step with the first rule so that `X` is instantiated to `s(0)` and the instantiated equation is reduced to `s(s(0)) = s(s(0))` which is trivially true. Hence we have found the solution `X`$\mapsto$`s(0)` to the given equation.

In order to be complete in general, *each* rule must be unified with *each* non-variable subterm of the given equation. This has the consequence that the narrowing method has a huge and infinite space even for simple programs. For instance, consider the previous rules for addition together with the following rules defining a sum function on naturals:

```
sum(0)     → 0
sum(s(N)) → s(N) + sum(N)
```

Then the narrowing method applied to the equation `sum(X) = s(0)` has an infinite search space due to the following infinite narrowing derivation (the instantiation of variables occurring in the equation is recorded at the derivation arrow):

$$
\begin{array}{lll}
\texttt{sum(X) = s(0)} & \leadsto_{\texttt{X}\mapsto\texttt{s(N1)}} & \texttt{s(N1)+sum(N1) = s(0)} \\
 & \leadsto_{\texttt{N1}\mapsto\texttt{s(N2)}} & \texttt{s(s(N2))+(s(N2)+sum(N2)) = s(0)} \\
 & \leadsto_{\texttt{N2}\mapsto\texttt{s(N3)}} & \cdots
\end{array}
$$

In order to reduce the search space, narrowing can be combined with rewriting (evaluation of a term): Before a narrowing step is applied, the equation is rewritten to normal form w.r.t. the given rules (thus this strategy is also called *normalizing narrowing*). This can avoid a lot of useless narrowing derivations. E.g., if we rewrite the second derived equation in the above example, we can immediately terminate this narrowing derivation:

$$\texttt{s(s(N2))+(s(N2)+sum(N2)) = s(0)} \ \to^* \ \texttt{s(s(N2+(s(N2)+sum(N2)))) = s(0)}$$

The last equation cannot be satisfied since the terms `s(N2+(s(N2)+sum(N2)))` and `0` are always different because there are no rules to reduce the symbols `s` and `0`. Hence we can terminate the unsuccessful narrowing derivation at this point.

The integration of rewriting into the narrowing process has at least two advantages:

1. The search space is reduced since useless narrowing derivations can be detected. As shown in [Fri85], [Han91] and [Han92] this has the consequence that functional-logic programs are more efficiently executable than the equivalent Prolog programs.
2. Deterministic computations are preferred. Since we assume a confluent and terminating set of rules, normal forms are unique and can be computed by any simplification strategy. Therefore rewriting can be implemented as a deterministic computation process in contrast to narrowing. Since rewriting is executed before a narrowing step is performed, the goal is computed in a deterministic way as long as possible. The preference of deterministic computations can save a lot of time and space as shown in [Han91].

But the integration of rewriting in narrowing derivations has also one disadvantage. Since the entire goal must be reduced to normal form after each narrowing step, the rewriting process may be costly. Hence we propose a useful optimization of the rewriting process: since the goal is in normal form before the narrowing step is applied and the narrowing step changes only small parts of the goal, rewriting can be restricted to a small number of positions in the narrowed goal in order to compute a new normal form. This optimization can speed up the execution mechanism of programming languages based on normalizing narrowing like SLOG [Fri85] or ALF [Han90, Han91].

Josephson and Dershowitz [JD89] have also presented an implementation of a narrowing strategy with rewriting. Although their main motivation was a space saving implementation by sharing common parts of different solutions in narrowing derivations, they have also presented an interesting technique to identify reducible subterms. Their technique is based on demons which are attached to subterms and wait for sufficient instantiation of their arguments. The correctness of their method is unclear since it depends on a particular firing strategy for the demons. Moreover, they generate a large number of demons because a demon is created for each potentially applicable rule at each subterm of the goal. These demons are not deleted even if the corresponding subterm has no connection to the goal (e.g., if the term `(X+Y)*0` is simplified to `0`, the demons corresponding to the subterm `(X+Y)` could be deleted). On the contrary, we will present an incremental rewriting algorithm which exactly implements a normalizing narrowing strategy. The overhead of this algorithm is quite small and the algorithm can be integrated in compiler-based implementations of functional-logic languages [Han91, Loo91].

In the next section we recall basic notions and results from term rewriting and narrowing. Our optimization techniques together with an incremental rewriting algorithm are presented in Section 3, and techniques for an efficient implementation of this algorithm are discussed in Section 4. Two extensions of this algorithm to non-linear rules and rules with nested functions on the left-hand side are shown in Section 5 and 6, respectively.

2 Preliminaries

In this section we recall basic notions of term rewriting [DJ90].

A *signature* is a set $\mathcal{F}$ of *function symbols*. Every $f \in \mathcal{F}$ is associated with an *arity*. 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. We call the terms $t_1,\ldots,t_n$ also *argument terms* of $f(t_1,\ldots,t_n)$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term is called *linear* if it does not contain multiple occurrences of the same variable. The function symbol heading term t is denoted by $\mathcal{H}ead(t)$.

The notion of *subterm* is defined through the notion of *position*. The set $\mathcal{O}(t)$ of *positions* in a term t is inductively defined by

$$\mathcal{O}(t) = \begin{cases} \{\Lambda\} & \text{if } t \in \mathcal{X} \\ \{\Lambda\} \cup \{i.\pi \mid 1 \leq i \leq n \text{ and } \pi \in \mathcal{O}(t_i)\} & \text{if } t = f(t_1,\ldots,t_n) \end{cases}$$

Hence positions are sequences of natural numbers. For the sake of readability we omit the last Λ in nonempty sequences of natural numbers, i.e., we write 1.2 instead of $1.2.\Lambda$. Positions are partially ordered by the prefix ordering $\leq$, i.e., $\pi \leq \pi'$ if there is a sequence ρ with $\pi \circ \rho = \pi'$. We write $\pi < \pi'$ if $\pi \leq \pi'$ and $\pi \neq \pi'$. Disjoint positions π, π' are denoted by $\pi \parallel \pi'$ (i.e., neither $\pi \leq \pi'$ nor $\pi' \leq \pi$). We call a position π *maximal* (in a set of positions) if there is no position π' with $\pi < \pi'$.

A position $\pi \in \mathcal{O}(t)$ denotes the *subterm* $t|_\pi$ of t, i.e.,

$$t|_\pi = \begin{cases} t & \text{if } \pi = \Lambda \\ t_i|_{\pi'} & \text{if } t = f(t_1, \ldots, t_n) \text{ and } \pi = i.\pi' \end{cases}$$

If $\pi \in \mathcal{O}(t)$, then $t[s]_\pi$ denotes the result of *replacing the subterm* $t|_\pi$ by the term s, i.e.,

$$t[s]_\pi = \begin{cases} s & \text{if } \pi = \Lambda \\ f(t_1, \ldots, t_i[s]_{\pi'}, \ldots, t_n) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } \pi = i.\pi' \end{cases}$$

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 *rewrite rule* $l \to r$ is a pair of terms l, r satisfying $l \notin \mathcal{X}$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ where l and r are called left-hand side and right-hand side, respectively. A rewrite rule is called *left-linear* if the left-hand side is linear, otherwise it is called *non-linear*. The *argument terms* of a rewrite rule are the argument terms of the left-hand side. A rewrite 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 rewrite rules. In the following we assume a given *term rewriting system* $\mathcal{R}$.

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 $\pi \in \mathcal{O}(t)$, a rewrite rule $l \to r$ and a substitution σ with $t|_\pi = \sigma(l)$ and $s = t[\sigma(r)]_\pi$. In this case we say t is *reducible* (at position π). A term t is called *irreducible* or in *normal form* if there is no term s with $t \to_{\mathcal{R}} s$. A substitution σ is called *irreducible* or *normalized* if $\sigma(x)$ is in normal form for all variables $x \in \mathcal{X}$.

$\to_{\mathcal{R}}^*$ denotes the transitive-reflexive closure of the rewrite relation $\to_{\mathcal{R}}$. $\mathcal{R}$ is called *terminating* if there are no infinite rewriting 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 $t_2 \to_{\mathcal{R}}^* t_3$. A terminating and confluent term rewriting system $\mathcal{R}$ is called *canonical*.

If $\mathcal{R}$ is canonical, we can decide the validity of an equation $s =_{\mathcal{R}} t$ (where $=_{\mathcal{R}}$ denotes validity w.r.t. the equations $\{l \doteq r \mid l \to r \in \mathcal{R}\}$) 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 $\pi \in \mathcal{O}(t)$ (i.e., $t|_\pi \notin \mathcal{X}$), a variant $l \to r$ of a rewrite rule and a substitution σ such that σ is a mgu of $t|_\pi$ and l and $t' = \sigma(t[r]_\pi)$. In this case we write $t \rightsquigarrow_{[\pi,l\to r,\sigma]} t'$ or simply $t \rightsquigarrow_{\sigma} t'$. For variables $x \in \mathcal{V}ar(t)$ with $\sigma(x) \neq x$ we say x is *bound* in the narrowing step $t \rightsquigarrow_{\sigma} t'$. If there is a narrowing sequence $t_1 \rightsquigarrow_{\sigma_1} t_2 \rightsquigarrow_{\sigma_2} \cdots \rightsquigarrow_{\sigma_{n-1}} t_n$, we write $t_1 \rightsquigarrow^*_{\sigma} t_n$ with $\sigma = \sigma_{n-1} \circ \cdots \circ \sigma_2 \circ \sigma_1$.

Narrowing is able to solve equations w.r.t. $\mathcal{R}$. For this purpose we introduce two new function symbols $=^?$ and *true* and add the rewrite rule $x =^? x \to true$ to $\mathcal{R}$. The following theorem states soundness and completeness of narrowing.

Theorem 1 ([Hul80]). *Let $\mathcal{R}$ be a canonical term rewriting system.*

1. *If $s =^? t \rightsquigarrow^*_{\sigma} true$, then $\sigma(s) =_{\mathcal{R}} \sigma(t)$.*
2. *If $\sigma'(s) =_{\mathcal{R}} \sigma'(t)$, then there exist a narrowing derivation $s =^? t \rightsquigarrow^*_{\sigma} true$ and a subsitution ϕ with $\phi(\sigma(x)) =_{\mathcal{R}} \sigma'(x)$ for all $x \in \mathcal{V}ar(s) \cup \mathcal{V}ar(t)$.*

Since this simple narrowing procedure (enumerating all narrowing derivations) has a huge search space, several authors have improved it by restricting the admissible narrowing derivations. For instance, Hullot [Hul80] has proved completeness of *basic narrowing* where a narrowing step in a position introduced by a substitution is forbidden. Fribourg [Fri85] has proposed *innermost narrowing* where narrowing is performed from innermost to outermost positions (this is only complete for totally defined functions), and Hölldobler [Höl89] has combined innermost and basic narrowing (which is complete also for partially defined functions).

Another improvement of simple narrowing is *normalizing narrowing* [Fay79] where the term is rewritten to 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 gives the same result (because $\mathcal{R}$ is canonical) whereas different narrowing steps may lead to different solutions and therefore all admissible narrowing steps must be considered. Hence in a sequential implementation rewriting can be efficiently implemented like reductions in functional languages whereas narrowing steps need costly backtracking management like in Prolog. For instance, if the equation $s =_{\mathcal{R}} t$ is valid, normalizing narrowing will prove it by a pure deterministic computation (reducing s and t to the same normal form) whereas simple narrowing would compute the normal form of s and t by costly narrowing steps.

Normalizing narrowing can also be combined with the other optimizations mentioned before. Rety [Ret87] has proved completeness of *normalizing basic narrowing*, Fribourg [Fri85] has proposed *normalizing innermost narrowing* and Hölldobler [Höl89] has combined innermost basic narrowing with normalization. Because of these important advantages, normalizing narrowing is the foundation of several programming languages which combines functional and logic programming like SLOG [Fri85] or ALF [Han91]. However, there is one problem with normalizing narrowing. Since the entire term or equation must be normalized before a narrowing step is applied, the normalization process may be very costly especially if only small parts of the term are influenced by the previous narrowing step. Therefore we present in the next section an incremental rewriting algorithm which computes the normal form of a term after a narrowing step and takes advantage of the fact that the term was in normal form before the application of the last narrowing step.

3 The incremental rewriting algorithm

In this section we will present an algorithm which computes the normal form of a term after a narrowing step and uses the fact that the narrowed term was in normal form. In the following we assume that the given term rewriting system $\mathcal{R}$ is canonical. Thus we can rewrite a term to normal form using an arbitrary strategy, and in the following we apply rewrite steps from innermost to outermost positions. Our method of optimizing the rewriting process after a narrowing step is based on two observations:

Starting condition: If the term t is in normal form and we apply a narrowing step at position π in t giving t', then, in order to compute the normal form of t', we have to rewrite the subterm $t'|_\pi$ (since the instantiated right-hand side of the applied rule may be not in normal form) and all subterms of t' where a variable of t has been bound in the narrowing step. All other subterms of t' (except those containing the subterms mentioned in the previous sentence) are in normal form since t was in normal form.

Stopping condition: Since we perform rewriting in an innermost manner, we try to apply a rewrite rule to a subterm after computing the normal form of its argument terms. But if no rewrite rule is applicable to any argument term, then we need not try to apply a rewrite rule to this subterm in order to compute the normal form after a narrowing step (this is only true under some additional conditions, see below).

The following example shows the optimized normalization process using these two conditions.

Example 1. The following rewrite rules are given:

```
f(a) → b
g(a) → b
h(a,X,b) → a
```

The term `h(f(X),g(f(Y)),f(g(X)))` is in normal form. If we apply the first rule to the subterm `f(X)` in a narrowing step, the variable `X` is bound to `a` and the narrowed term is `h(b,g(f(Y)),f(g(a)))`. The starting condition tells us that we must rewrite the first argument `b` (the narrowed subterm) and the subterm `g(a)` (since here a variable has been bound). Rewriting `g(a)` gives `h(b,g(f(Y)),f(b))`, but there is no rewrite rule applicable to the subterms `b` and `f(b)`. Hence, by the stopping condition, we can terminate the normalization process without trying to apply rewrite rules to the subterms `f(Y)`, `g(f(Y))` and `h(b,g(f(Y)),f(b))`. This can save a lot of unnecessary rewrite attempts if the second argument is a large term (instead of `g(f(Y))`) or the term is embedded in a larger one. □

The starting condition is justified by the following lemma.

Lemma 2 (Starting lemma). *Let t be a term in normal form, $x \in \mathcal{V}ar(t)$ and $\sigma = \{x \mapsto t'\}$ be a substitution with t' in normal form. If $\sigma(t)$ is reducible at position π, then $\pi \in \mathcal{O}(t)$ and $x \in \mathcal{V}ar(t|_\pi)$.*

Proof. Let $\sigma(t)$ be reducible at position π. Assume $\pi \notin \mathcal{O}(t)$. Since $\pi \in \mathcal{O}(\sigma(t))$ and $\sigma(t)$ reducible at position π, t' must be reducible in contradiction to our assumptions. Therefore $\pi \in \mathcal{O}(t)$.

Now assume that $x \notin \mathcal{V}ar(t|_\pi)$. Since σ replaces only variable x, we have $t|_\pi = \sigma(t)|_\pi$, i.e., t is also reducible at position π in contrast to our assumptions. Therefore $x \in \mathcal{V}ar(t|_\pi)$. $\square$

The lemma can easily be extended to substitutions which replace more than one variable. Note that if the substituted term t' is not in normal form, then $\pi \in \mathcal{O}(t)$ is not implied by the fact that $\sigma(t)$ is reducible at position π since the reduction may be performed in the substituted term t'. But the important consequence of this lemma is that rewriting in a subterm is unnecessary if this subterm is not influenced by the narrowing substitution.

The stopping condition expresses the fact that we can stop the rewriting process in one path of the term if no rewrite rule is applicable at a distinct position. The next lemma shows that this is true under additional conditions.

Lemma 3 (Stopping lemma). *Let t and s be terms in normal form, $s \notin \mathcal{X}$, $\pi \in \mathcal{O}(t)$ with $t|_\pi \in \mathcal{X}$ and all rewrite rules be left-linear. If the head symbol $\mathcal{H}ead(s)$ does not occur in an argument term of any rewrite rule, then $t[s]_\pi$ is in normal form.*

Proof. Assume that $t' = t[s]_\pi$ is not in normal form. Then there is a rule $l \to r$, a position $\pi' \in \mathcal{O}(t')$ and a substitution σ with $t'|_{\pi'} = \sigma(l)$. We distinguish the following cases:

1. $\pi \leq \pi'$, i.e., there exists ρ with $\pi' = \pi \circ \rho$: Then $\sigma(l) = t'|_{\pi'} = t'|_{\pi\circ\rho} = s|_\rho$. Hence s is reducible which contradicts our assumption.
2. $\pi' < \pi$, i.e., $\pi = \pi' \circ \rho$ for some nonempty ρ: From $t'|_{\pi'} = \sigma(l)$ we can infer $\sigma(l)|_\rho = s$. Since the head symbol $\mathcal{H}ead(s)$ does not occur in an argument of l, s must belong to the substitution σ, i.e., there exists $y \in \mathcal{V}ar(l)$ with $\sigma(y) = \cdots s \cdots$. Now replace s by the variable $t|_\pi$ in $\sigma(y)$ and denote this new substitution by σ'. Then $t|_{\pi'} = \sigma'(l)$ since l is linear. Hence t is reducible in contrast to our assumption.
3. $\pi \parallel \pi'$: Then $t|_{\pi'} = t'|_{\pi'} = \sigma(l)$ since t and t' differ only at position π. Hence t is reducible in contrast to our assumption.

Therefore our assumption is wrong and thus $t[s]_\pi$ must be in normal form. $\square$

This lemma is not true if a rewrite rule is not left-linear. For instance, if there is the rule

```
f(Y,Y) → Y
```

then `f(1,X)` and `1` are terms in normal form, but `f(1,1)` is not in normal form. Fortunately, most functional programs are written with the left-linearity condition, but we will also discuss a solution for general term rewrite rules in Section 5.

The relation between the stopping lemma and the stopping condition will be clarified by the following lemma.

Lemma 4. *Let t be a term in normal form, $\pi \in \mathcal{O}(t)$, x a variable and all rewrite rules be left-linear. Then $t[x]_\pi$ is also in normal form.*

Proof. Assume that $t' = t[x]_\pi$ is not in normal form. Then there is a rule $l \to r$, a position $\pi' \in \mathcal{O}(t')$ and a substitution σ with $t'|_{\pi'} = \sigma(l)$. We distinguish the following cases:

1. $\pi \leq \pi'$: Then $\pi = \pi'$ since $t'|_\pi$ is a variable. Because $t'|_{\pi'} = x$, l must be a variable which contradicts our assumptions about rewrite rules (otherwise, if we allow rewrite rules $l \to r$ with $l \in \mathcal{X}$, this rule would also be applicable to t at position π).
2. $\pi' < \pi$, i.e., $\pi = \pi' \circ \rho$ for some nonempty ρ: Because $t'|_\pi = x$ and l is a linear term, we can modify σ to σ' such that $\sigma'(l) = t|_{\pi'}$ (replace one occurrence of x by $t|_\pi$ in the codomain of σ). Hence t is reducible in contrast to our assumption.
3. $\pi \parallel \pi'$: Since t and t' differ only at position π, $t|_{\pi'} = t'|_{\pi'} = \sigma(l)$. Hence t is reducible in contrast to our assumption.

□

Lemma 3 and Lemma 4 imply the correctness of the stopping condition: if a subterm is in normal form and the term "outside" the subterm is also in normal form, then the entire term is in normal form provided that all rules are left-linear and do not contain the head symbol of the subterm in an argument term. The last restriction seems to be very strong for general term rewriting systems. But in most functional logic languages (SLOG [Fri85], K-LEAF [BGL$^+$87], BABEL [MR92], ALF [Han90] etc.) a distinction is made between function symbols to construct data terms (called *constructors*) and function symbols to operate on data terms (called *defined functions*). This is also the case for pure functional languages like ML [HMM86] or Miranda [Tur85]. The partition of the set of functions symbols into constructors and defined functions comes with a restriction on the set of rules:

> (*) Each rule must define a unique function, i.e., the left-hand side of the rule must contain a defined function as the head symbol and all argument terms contain variables and constructors but no defined function.[1]

With this restriction it is clear that a subterm is reducible only if it contains a defined function at the top. Therefore positions with constructor symbols need not be considered in the rewriting process (this is essential for an efficient implementation of such languages [Han91]). In this situation the condition on occurrences of function symbols in argument terms (in the stopping lemma) is no real restriction: since we try to apply rewrite rules only to subterms with a defined function symbol at the top, the occurrence condition of Lemma 3 is always satisfied if the program satisfies condition (*).

For the rest of this section we assume that the set of all function symbols is divided into a set $\mathcal{C}$ of constructors and a set $\mathcal{D}$ of defined function symbols and all

[1] We do not consider equations between constructor terms which are admissible in some languages.

rewrite rules satisfy condition (*) (see Section 6 for a relaxation of this requirement). The following algorithm describes the optimized rewriting process after a narrowing step. We assume that rewriting is performed using an innermost strategy, but this is not essential since we can use any strategy because of the canonicity of the term rewriting relation. In the description we use a set O of positions to describe the subterm positions where we have to try to apply a rewrite rule. The set is initialized in the narrowing step and manipulated during the rewriting process.

Algorithm: Perform a narrowing step and compute the normal form

Input: Term t in normal form, narrowing position π_0 and rule $l_0 \rightarrow r_0$ such that $t|_{\pi_0}$ and l_0 are unifiable with mgu σ_0

Output: Normal form of $\sigma_0(t[r_0]_{\pi_0})$

1. Compute the mgu σ_0 of $t|_{\pi_0}$ and l_0
2. $t' := \sigma_0(t[r_0]_{\pi_0})$
3. $O := funpos(t', \pi_0)$
4. **if** $\mathcal{H}ead(t'|_{\pi_0}) \in \mathcal{C}$ **then** $O := O \cup father(t', \pi_0)$
5. $B := \{\pi \in \mathcal{O}(t) \mid \mathcal{H}ead(t|_\pi) \in \mathcal{D}$ and $t|_\pi$ contains variables bound by σ_0 and π maximal with this property$\}$
6. $O := O \cup \bigcup_{\pi \in B} funpos(t', \pi)$
7. **while** $O \neq \emptyset$
   do let π be a maximal position in O
   $O := O - \{\pi\}$
   if there is a rewrite rule $l \rightarrow r$ and substitution σ with $t'|_\pi = \sigma(l)$
   then $t' := t'[\sigma(r)]_\pi$
   $O := O \cup funpos(t', \pi)$
   if $\mathcal{H}ead(t'|_\pi) \in \mathcal{C}$ **then** $O := O \cup father(t', \pi)$

where $funpos(t, \pi) = \{\pi_t \in \mathcal{O}(t) \mid \pi \leq \pi_t \text{ and } \mathcal{H}ead(t|_{\pi_t}) \in \mathcal{D}\}$

and $$father(t, \pi) = \begin{cases} \emptyset & \text{if } \pi = \Lambda \\ \{\pi'\} & \text{if } \pi = \pi'.n \text{ and } \mathcal{H}ead(t|_{\pi'}) \in \mathcal{D} \\ father(t, \pi') & \text{if } \pi = \pi'.n \text{ and } \mathcal{H}ead(t|_{\pi'}) \in \mathcal{C} \end{cases}$$

The first two steps apply the narrowing rule to the given term. Step 3 initializes the set O with the positions of the instantiated right-hand side of the applied rule. The addition of the father position in step 4 is necessary since the rewriting process can only be terminated if the normalized subterm has a defined function symbol at the top (by condition (*) and Lemma 3). Step 5 computes the potential rewrite positions by Lemma 2, i.e., the subterm positions where a variable has been bound. Note that only the maximal positions are stored since the smaller positions (nearer to the root) are added during rewriting by the function *father*. Step 6 adds all occurrences of defined functions in these subterms. This is necessary because the unifier σ_0 is not normalized in general. If we use a narrowing strategy which ensures that the narrowing substitutions are always normalized like the innermost strategy of SLOG [Fri85], then we can simplify step 6 to the assignment

$$O \ := \ O \ \cup \ B$$

Step 7 describes the rewriting process where rewrite rules are only applied at positions of the restricted set O. The choice of a *maximal* position in the rewriting loop ensures that rewriting is performed in an innermost manner. If a rewrite rule can be applied and the instantiated right-hand side has a constructor at the top, the next outermost position is added to O since in this case we cannot terminate the rewriting process by Lemma 3.

The correctness of this algorithm follows from the previous lemmas:

Theorem 5. *If all rewrite rules are left-linear and satisfy condition (*), then the above algorithm is correct, i.e., it computes the normal form after the narrowing step.*

Proof. Before rewriting is started, the position set O is initialized with the subterm positions of the instantiated right-hand side of the narrowing rule and the positions where a variable has been bound. Rewriting is not possible at other positions by the previous lemmas. Note that not all positions from the root to the subterms are stored (which must be correctly done by Lemma 2) but only the maximal positions: the "father" of a position is only added to O if a rewrite step is possible and yields a constructor symbol at this position. Otherwise, it is not necessary to try rewriting at the father position by Lemma 3 and Lemma 4. □

Example 2. We show a computation of the algorithm w.r.t. the rewrite rules of Example 1, i.e., the constructors are $\mathcal{C} = \{\mathtt{a}, \mathtt{b}\}$ and the defined functions are $\mathcal{D} = \{\mathtt{f}, \mathtt{g}, \mathtt{h}\}$. The term $t = \mathtt{h(g(f(X)),g(f(Y)),f(g(X)))}$ is in normal form w.r.t. these rules. We perform a narrowing step with rule $\mathtt{f(a)}\to\mathtt{b}$ at position 1.1 (subterm `f(X)`).

- The mgu of $t/1.1$ and `f(a)` binds `X` to `a`.
- The narrowed term is $t' = \mathtt{h(g(b),g(f(Y)),f(g(a)))}$.
- The position set computed after step 6 is $O = \{1, 3.1\}$, i.e., the terms `g(b)` and `g(a)` (since the arguments of these terms have changed).
- No rewrite rule is applicable at position 1: $O = \{3.1\}$.
- Rewrite rule $\mathtt{g(a)}\to\mathtt{b}$ is applicable at position 3.1: $t' = \mathtt{h(g(b),g(f(Y)),f(b))}$ and $O = \{3\}$
- No rule is applicable at position 3: $O = \emptyset$.

Now we have computed the new normal form `h(g(b),g(f(Y)),f(b))`. Note that without our optimizations we would also try to apply all rewrite rules at positions 2.1, 2 and Λ. □

We cannot state a general result for the efficiency improvement of our optimizations because this strongly depends on the selected examples. In some cases there is no improvement (if the binding of variables has the effect that all defined function symbols can be rewritten) where in other cases we may have a dramatic improvement (for instance, if `Y` is replaced by a large term containing function symbols but not the variable `X`). But we want to remark that the overhead introduced by this incremental rewriting technique is quite small (only during variable binding in a narrowing step we have to store new information in comparison to a complete innermost rewriting derivation). For some typical so-called *generate-and-test* programs (like permutation sort [Fri85] or generation of mobiles [Han92]) where narrowing generates a part of the solution and rewriting tests whether it is a solution we can avoid up to 70% of unnecessary rewriting attempts.

4 Implementation of the incremental rewriting algorithm

The reader may be under the impression that the implementation of our incremental rewriting algorithm is too complex for an efficient execution in a functional logic language. But this depends on the chosen narrowing strategy and therefore we will give some hints for an efficient implementation of our algorithm.

First of all the function $funpos$ seems to be costly because it requires a search through instantiated subterms after each narrowing or rewriting step (in step 3, 6 and 7 in the algorithm). We can avoid this dynamic search by using a narrowing strategy which ensures that rewriting inside narrowing substitutions is not performed. For instance, the innermost strategy of SLOG [Fri85] and the innermost basic strategy [Höl89] of ALF [Han91] have this property.[2] In this case the compiler can determine all positions of defined function symbols in the right-hand side of a rewrite rule [Han91] in step 3 and 7, and step 6 simplifies to $O := O \cup B$. Hence we can avoid the dynamic search in the $funpos$ function calls.

Secondly, it seems to be disturbing that the algorithm is based on the management of an explicit set O of rewriting positions. But this causes no overhead in practice since the implementation of the functional logic language ALF [Han91] is based on an explicit stack for storing positions. Moreover, this explicit management of positions is the key for a simple but efficient implementation based on an extension of the Warren Abstract Machine [War83]. Hence our incremental rewriting algorithm can be implemented on the basis of the ALF implementation if the structure of the position stack is modified such that the $father$ function is efficiently computable (in the current implementation there is no path from "son" to "father" nodes in the term representation).

Hence the only remaining critical operation is the computation of positions where a variable has been bound in the narrowing step (set B in step 5). This operation is close to the Prolog-II predicate `freeze` and therefore we can use a similar implementation technique [Boi86]. The predicate `freeze(X,G)` delays the execution of the goal `G` until the variable `X` is bound to a non-variable term. This behaviour can

[2] In the innermost basic strategy the narrowing substitutions are not normalized in general. If the goal is reducible inside a narrowing substitution, we can safely cut this derivation without loosing completeness (Rety's SL-test [Ret87]). But a narrowing strategy without this SL-test can be efficiently implemented in a compiler-based system [Han90, Han91]. Therefore rewriting is never performed inside narrowing substitutions in ALF.

be implemented by connecting the variable `X` to the list of goals `G` which should be executed if `X` is instantiated. The unification procedure must be modified such that the goal list associated to a frozen variable `X` is executed if `X` is bound to a non-variable term [Boi86]. A similar technique can be used to implement step 5 of our incremental rewriting algorithm. For each goal variable x we associate the position set

$$B(x) := \{\pi \in \mathcal{O}(t) \mid \mathcal{H}ead(t|_\pi) \in \mathcal{D}, x \in \mathcal{V}ar(t|_\pi), \pi \text{ maximal with this property}\} \ .$$

This set can be simply built up during the construction of a new term and updated during rewriting and unification of terms. If a variable x is bound to a term in a narrowing step, we change the binding algorithm so that the associated positions $B(x)$ are added to the current position set O. This implements steps 5 and 6 of our algorithm.

Note that this implementation is very similar to Naish's proposal [Nai91] for translating function definitions into NU-Prolog predicates with particular "when" declarations. The "when" declarations ensure that a function call is evaluated when the arguments are sufficiently instantiated. Hence Naish's method implements a normalization process for a restricted class of rewrite rules. Since Naish does never instantiate goal variables in an application of a rewrite rule (i.e., narrowing is not included in his proposal), his operational semantics is incomplete w.r.t. the standard declarative semantics for Horn clause logic with equality in contrast to our approach.

There is one pitfall in our implementation. In contrast to the usage of `freeze` in Prolog-II or when declarations in NU-Prolog, in our case it is possible that some part of the goal may be deleted during the normalization process if a rewrite rule $l \to r$ with $\mathcal{V}ar(r) \neq \mathcal{V}ar(l)$ is applied. For instance, the application of the rule `X*0`$\to$`0` to the term `(Y+2)*0` deletes the subterm `(Y+2)`. From an operational point of view this is a positive effect in comparison to pure logic languages because such deleting rules reduce the search space (narrowing the subterm `(Y+2)` is no longer necessary). But from an implementational point of view such deleting rules cause a complication: we must update the set $B(x)$ for each variable x occurring in a deleted subterm (otherwise the set O will be inconsistent). Consequently, for deleting rules the compiler must generate additional instructions which updates the position sets associated to variables occurring in the deleted terms.

5 Non-linear rewrite rules

If one of the rules of the given term rewriting system $\mathcal{R}$ has a non-linear left-hand side, the stopping condition of Section 3 cannot be applied. In such a case we can only restrict the set of positions where we have to start the rewriting process (Lemma 2), but then we have to try all positions up to the root even if the innermost subterms are in normal form.

Example 3. Consider the following term rewriting system

```
f(X,X) → b
```

```
g(a)    → a
h(b)    → b
```

and the irreducible term

```
f(h(h(h(g(Y)))),h(h(h(Y)))) .
```

We can apply a narrowing step at position 1.1.1.1 using the second rule. This narrowing step binds variable `Y` to `a` and yields the term `f(h(h(h(a))),h(h(h(a))))`. Although the subterms `a` and `h(a)` are in normal form, we cannot terminate the rewriting process since an application of the first rewrite rule at position Λ yields the term `b` in normal form. □

In order to avoid such problems we may restrict ourselves to left-linear rules as done in other functional logic languages like K-LEAF [BGL+87] or BABEL [MR92]. But we can also handle non-linear rules by a simple extension of our incremental rewriting algorithm. Since the only positions where a rewrite rule is applicable in contrast to the stopping condition are positions with an applicable non-linear rewrite rule (see proof of Lemma 3), we simply add these "problematic" positions to the set O in the algorithm. For this purpose we call a function symbol f *non-linear* if there is a rewrite rule $l \to r$ with $\mathcal{H}ead(l) = f$ and l is not linear. Now we add after step 6 in the incremental rewriting algorithm the following extension of the set O:

$$O := O \cup \{\pi \in \mathcal{O}(t') \mid \mathcal{H}ead(t'|_\pi) \text{ is a non-linear function symbol}\}$$

i.e., we simply add the positions with non-linear functions to O. This has the effect that rewriting is also tried at these positions.

Example 4. Consider the term rewriting system of Example 3 and the term

```
f(h(h(h(g(Y)))),h(h(h(Y)))) .
```

After the narrowing step at position 1.1.1.1 (giving the term `f(h(h(h(a))), h(h(h(a))))`) the set O is initialized to

$$O = \{\Lambda, 1.1.1, 2.1.1\}$$

by the modified algorithm. Since the term is irreducible at positions 1.1.1 and 2.1.1, a rewrite rule is applied at the remaining position Λ which is successful and yields the term `b`. □

6 Rules with nested functions symbols

In Section 3 we have required the distinction between constructors and defined function symbols in order to use the stopping condition. In the rewrite rules we have the restriction (*) that argument terms do not contain defined function symbols. Although this restriction seems to be reasonable and is also used in many functional logic languages like K-LEAF [BGL+87], SLOG [Fri85] or BABEL [MR92], in some cases it is useful to have general rewrite rules which are only used to compute the normal form of a term (therefore the language ALF [Han91] allows such rules). For instance, Fribourg [Fri85] has argued that rewriting with inductive axioms can reduce the search space and is justified if someone is interested in ground-valid answers (i.e., answers which are valid for each ground substitution applied to it). A typical inductive axiom is `rev(rev(L)) = L` if `rev` denotes the function which reverses a list. If we allow rules which do not satisfy restriction (*), the stopping condition cannot be applied as in the incremental rewriting algorithm.

Example 5. Consider the rewrite rule

`f(g(a))` $\to$ `a`

(where `f` and `g` are defined function symbols) and a narrowing step where the rule `h(a)`$\to$`a` is applied to the term `k(h(X),f(g(X)))` giving the term `k(a,f(g(a)))`. Suppose that the term `g(a)` is irreducible. In this case our incremental rewriting algorithm will not try to reduce the subterm `f(g(a))`. However, this must be done in order to compute the normal form of the entire term. □

If we want to handle rules with defined function symbols in argument terms, it is possible to extend our incremental rewriting algorithm to do this. The essential idea is to add positions to the set O if the current subterm is irreducible but has a head symbol which occurs in an argument term of a rewrite rule. For this purpose we define the *level* of a defined function symbol f as the set of position depths where f occurs in an argument term of a rule. To be more precise,

$$level(f) = \{depth(\pi) \mid \text{there is a rule } l \to r \text{ and position } \pi \neq \Lambda \text{ with } \mathcal{H}ead(l|_\pi) = f\}$$

where $depth(\pi) = \begin{cases} 0 & \text{if } \pi = \Lambda \\ 1 + depth(\pi') & \text{if } \pi = \pi'.n \end{cases}$

If a subterm $t|_\pi$ of a term is in normal form but there exists $i \in level(\mathcal{H}ead(t|_\pi))$, then the entire term t may be reducible at the i-th ancestor of π. Thus we must add this position to the position set O. Hence we define the i-th ancestor of a position π by

$$anc(i,\pi) = \begin{cases} \pi & \text{if } i = 0 \\ anc(i-1,\pi') & \text{if } i > 0 \text{ and } \pi = \pi'.n \\ \textit{undefined} & \text{if } i > 0 \text{ and } \pi = \Lambda \end{cases}$$

and add the following else-branch to the first if-statement in step 7 in the incremental rewriting algorithm:

else $O := O \cup \{anc(i,\pi) \mid i \in level(\mathcal{H}ead(t'|_\pi)) \text{ and } anc(i,\pi) \text{ is defined}\}$

Example 6. Consider Example 5 where `k(a,f(g(a)))` is the term after the narrowing step and $O = \{\Lambda, 2.1\}$. Since this term is irreducible at position 2.1 and $level(g) = \{1\}$, our modified incremental rewriting algorithm extends the position set O by the position $2 = anc(1, 2.1)$. Thus the next rewriting step is performed at position 2 (subterm `f(g(a))`) which is necessary to compute the normal form of the entire term. □

7 Conclusions

We have presented a useful optimization for functional logic languages based on narrowing with normalization. This optimization is based on the observation that rewriting can be restricted to a small number of positions after a narrowing step since the term was in normal form before the narrowing step has been applied. We have given two sufficient criteria for the optimization: the starting condition restricts the number of term positions where the rewriting process is initiated, and the stopping condition yields a criterion for the early termination of the rewriting process. Our presented incremental rewriting algorithm combines both conditions. We have also discussed techniques for an efficient implementation of the algorithm and extensions to deal with rewrite rules which have non-linear left-hand sides and nested function symbols on the left-hand side.

The presented algorithm is not optimal in the sense that it avoids all unnecessary rewriting attempts. For instance, consider the following term rewriting system

```
f(c(X)) → X
g(c(a)) → a
```

and the narrowing step

$$\mathtt{h(f(Y),g(Y))} \leadsto_{[1,\ \mathtt{f(c(X))}\to\mathtt{X},\ \{\mathtt{Y}\mapsto\mathtt{c(X)}\}]} \mathtt{h(X,g(c(X)))}\ .$$

Now our incremental rewriting algorithm attempts to apply a rewrite rule at position 2 since `Y` has been bound to `c(X)` in the subterm `g(Y)`. But a rewrite rule is not applicable at this position because the argument term `c(X)` is not sufficiently instantiated. To avoid unnecessary rewrite attempts of this kind one can implement rewrite rules as demons waiting for sufficient instantiation of the arguments of a subterm [JD89]. In order to reduce the number of demons the rules may be translated into rules with a uniform structure on the left-hand side [MKLR90] so that only one demon is attached to each potentially reducible subterm. An integration of such techniques in a compiler-based implementation of a functional-logic language is an interesting topic for future research.

Acknowledgements. The author is grateful to Michael Gollner, Rudolf Opalla and Andreas Werner for interesting discussions and their comments on this paper.

References

[BGL+87] P.G. Bosco, E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. A complete semantic characterization of K-LEAF, a logic language with partial functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pp. 318–327, San Francisco, 1987.

[BL86] M. Bellia and G. Levi. The Relation between Logic and Functional Languages: A Survey. *Journal of Logic Programming (3)*, pp. 217–236, 1986.

[Boi86] P. Boizumault. A general model to implement dif and freeze. In *Proc. Third International Conference on Logic Programming (London)*, pp. 585–592. Springer LNCS 225, 1986.

[DJ90] 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.

[DL86] D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations, and Equations.* Prentice Hall, 1986.

[Fay79] 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.

[Fri85] 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.

[Han90] 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.

[Han91] M. Hanus. Efficient Implementation of Narrowing and Rewriting. In *Proc. Int. Workshop on Processing Declarative Knowledge*, pp. 344–365. Springer LNAI 567, 1991.

[Han92] M. Hanus. Improving Control of Logic Programs by Using Functional Logic Languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming.* Springer LNCS, 1992.

[HMM86] R. Harper, D.B. MacQueen, and R. Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, University of Edinburgh, 1986.

[Höl89] S. Hölldobler. *Foundations of Equational Logic Programming.* Springer LNCS 353, 1989.

[Hul80] J.-M. Hullot. Canonical Forms and Unification. In *Proc. 5th Conference on Automated Deduction*, pp. 318–334. Springer LNCS 87, 1980.

[JD89] A. Josephson and N. Dershowitz. An Implementation of Narrowing. *Journal of Logic Programming (6)*, pp. 57–77, 1989.

[Loo91] R. Loogen. From Reduction Machines to Narrowing Machines. In *Proc. of the TAPSOFT '91*, pp. 438–457. Springer LNCS 494, 1991.

[MKLR90] J.J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy Narrowing in a Graph Machine. In *Proc. Second International Conference on Algebraic and Logic Programming*, pp. 298–317. Springer LNCS 463, 1990.

[MR92] 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.

[Nai91] L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 15–26. Springer LNCS 528, 1991.

[Ret87] P. Rety. Improving basic narrowing techniques. In *Proc. of the Conference on Rewriting Techniques and Applications*, pp. 228–241. Springer LNCS 256, 1987.

[Sla74] 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.

[Tur85] D. Turner. Miranda: A non-strict functional language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture, Nancy, France*, pp. 1–16. Springer LNCS 201, 1985.

[War83] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Stanford, 1983.