In *Proc. 5th International Conference on Compiler Construction*
Edinburgh, April 1994, Springer LNCS 786, pp. 68–82

Towards the Global Optimization of Functional Logic Programs*

Michael Hanus

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

Abstract. Functional logic languages amalgamate functional and logic programming paradigms. They can be efficiently implemented by extending techniques known from logic programming. In this paper we show how global information about the call modes of functions can be used to optimize the compilation of functional logic programs. Since mode information has been successfully used to improve the implementation of pure logic programs and these techniques can be applied to implementations of functional logic programs as well, we concentrate on optimizations which are unique to the operational semantics of functional logic programs. We define a suitable notion of modes for functional logic programs and present compile-time techniques to optimize the normalization process during the execution of functional logic programs.

1 Introduction

In recent years, a lot of proposals have been made to amalgamate functional and logic programming languages [7, 17]. Functional logic languages with a sound and complete operational semantics are based on narrowing (e.g., [10, 12, 26, 28]), a combination of the reduction principle of functional languages and the resolution principle of logic languages. Narrowing, originally introduced in automated theorem proving [29], 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`:

```
   0 + N → N              (R1)
s(M) + N → s(M + N)       (R2)
```

The equation `X+s(0)=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 `s(s(0))=s(s(0))` which is trivially true. Hence we have found the solution `X`$\mapsto$`s(0)` to the given equation. □

* The research described in this paper was supported by the German Ministry for Research and Technology (BMFT) under grant ITS 9103. The responsibility for the contents of this publication lies with the author.

In order to ensure completeness in general, *each* rule must be unified with *each* non-variable subterm of the given equation which yields a huge search space. This situation can be improved by particular narrowing strategies which restrict the possible positions for the application of the next narrowing step (see [17] for a detailed survey). In this paper we are interested in an *innermost narrowing* strategy where a narrowing step is performed at the leftmost innermost position. This corresponds to eager evaluation in functional languages.

However, the restriction to particular narrowing positions is not sufficient to avoid a lot of useless derivations since the uncontrolled instantiation of variables may cause infinite loops. For instance, consider the rules in Example 1 and the equation `(X+Y)+Z=0`. Applying innermost narrowing to this equation using rule R_2 produces the following infinite derivation (the instantiation of variables occurring in the equation is recorded at the derivation arrow):

$$\begin{array}{rl} \texttt{(X+Y)+Z = 0} \leadsto_{\{\texttt{X}\mapsto\texttt{s(X1)}\}} & \texttt{s(X1+Y)+Z = 0} \\ \leadsto_{\{\texttt{X1}\mapsto\texttt{s(X2)}\}} & \texttt{s(s(X2+Y))+Z = 0} \\ \leadsto_{\{\texttt{X2}\mapsto\texttt{s(X3)}\}} & \cdots \end{array}$$

To avoid such useless derivations, narrowing can be combined with simplification (evaluation of a term): Before a narrowing step is applied, the equation is rewritten to normal form w.r.t. the given rules [9, 10] (thus this strategy is also called *normalizing narrowing*). The infinite narrowing derivation above is avoided by rewriting the first derived equation to normal form:

$$\texttt{s(X1+Y)+Z = 0} \rightarrow \texttt{s((X1+Y)+Z) = 0}$$

The last equation can never be satisfied since the terms `s((X1+Y)+Z)` and `0` are always different due to the absence of rules for the symbols `s` and `0`. Hence we can safely terminate the unsuccessful narrowing derivation at this point. The integration of rewriting into narrowing derivations has the following advantages:

1. The search space is reduced since useless narrowing derivations can be detected. As a consequence, functional logic programs are more efficiently executable than equivalent Prolog programs [10, 13, 14].[2]
2. There is a preference for deterministic computations. Since we assume a confluent and terminating set of rules, normal forms are unique and can be computed by any simplification strategy. Hence normalization can be deterministically implemented. Since rewriting is executed before each nondeterministic narrowing step, 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 [13].

Therefore we consider in this paper a *normalizing innermost narrowing* strategy where the computation of the normal form between narrowing steps is performed by applying rewrite rules from innermost to outermost positions, i.e., a rewrite rule is applied to a term only if each of its subterms is in normal form. Such an operational semantics can be efficiently implemented by extending compilation techniques known from logic programming [12, 13].

[2] It is easy to see that the Prolog program corresponding to the above example would run into an infinite loop.

The integration of normalization into narrowing derivations has also one disadvantage. Since the entire goal must be reduced to normal form after each narrowing step, the normalization process may be costly. Fortunately, it is possible to normalize the terms in an incremental manner [15] since normalization steps after a narrowing step can only be performed at positions where some variables have been instantiated. However, better optimizations could be performed if the evaluation modes for functions are known at compile time. In this paper we define the notion of evaluation modes, which is different from logic programs [35], and show possible compile-time optimizations using these modes. We are not interested in low-level code optimizations to improve primitive unification instructions since such techniques, which have been developed for pure logic programs (e.g., [24, 25, 31, 32, 33, 34, 35]), can be applied to functional logic programs as well due to the similarities between WAM-based Prolog implementations and implementations of functional logic languages [12, 13, 23]. We limit our discussion to optimizations which are unique to functional logic programs based on an eager evaluation strategy like ALF [12, 13], LPG [1], or SLOG [10]. The automatic derivation of mode information for functional logic programs is a different topic which will be addressed in a forthcoming paper [18].

After a precise definition of the operational semantics in Section 2, we define the notion of modes for functional logic programs in Section 3. Section 4 discusses the optimization techniques using particular mode information. Experimental results for these optimization techniques are presented in Section 5, and some peculiarities of the automatic mode derivation for functional logic programs are discussed in Section 6.

2 Normalizing narrowing

To define the operational semantics considered in this paper in a precise way, we recall basic notions of term rewriting [8].

A *signature* is a set $\mathcal{F}$ of *function symbols*. 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. We denote by $\mathcal{T}(\mathcal{F},\mathcal{X})^n$ the set $\{\langle t_1,\ldots,t_n\rangle \mid t_i \in \mathcal{T}(\mathcal{F},\mathcal{X}), i = 1,\ldots,n\}$ of n-tuples of terms ($n \geq 0$). The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term t is called *ground* if $\mathcal{V}ar(t) = \varnothing$.

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 [12], BABEL [26], K-LEAF [11], SLOG [10]). Hence we assume that the signature $\mathcal{F}$ is partitioned into two sets $\mathcal{F} = \mathcal{C} \cup \mathcal{D}$ with $\mathcal{C} \cap \mathcal{D} = \varnothing$. A *constructor term* t is built from constructors and variables, i.e., $t \in \mathcal{T}(\mathcal{C},\mathcal{X})$. An *innermost term* t [10] is an operation applied to constructor terms, i.e., $t = f(t_1,\ldots,t_n)$ with $f \in \mathcal{D}$ and $t_1,\ldots,t_n \in \mathcal{T}(\mathcal{C},\mathcal{X})$. A *function call* $f(t_1,\ldots,t_n)$ is an operation $f \in \mathcal{D}$ applied to arbitrary terms. Such a term is also called *f-rooted term*.

A (*rewrite*) *rule* $l \to r$ is a pair of an innermost term l and a term r satisfying $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ where 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.[4] In the following we assume a given *term rewriting system* $\mathcal{R}$.

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 [8] for details).

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 $t_2 \to^*_{\mathcal{R}} t_3$. A terminating and confluent term rewriting system $\mathcal{R}$ is called *convergent*.

If $\mathcal{R}$ is convergent, 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 p in t (i.e., $t|_p \notin \mathcal{X}$), a variant $l \to r$ of a rewrite rule and 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 $t \rightsquigarrow_\sigma t'$. 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 a new operation symbol `=` and a new constructor `true` and add the rewrite rule

[3] For the sake of simplicity we consider only unconditional rules, but our results can easily be extended to conditional rules.

[4] We will apply rules in two ways: (a) in rewrite steps to evaluate terms, and (b) in narrowing steps to solve equations. Therefore we will sometimes distinguish between *rewrite rules* and *narrowing rules*. Usually, the set of rewrite rules and the set of narrowing rules are identical, but in some languages it is also possible to use some rules only for rewrite steps or only for narrowing steps (e.g., in ALF [12, 13] or SLOG [10]).

x=$x \to$ `true` to $\mathcal{R}$. Then the following theorem states soundness and completeness of narrowing.

Theorem 1 [20]. *Let $\mathcal{R}$ be a convergent 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 substitution ϕ with $\phi(\sigma(x)) =_\mathcal{R} \sigma'(x)$ for all $x \in \mathcal{V}ar(s) \cup \mathcal{V}ar(t)$.*

Thus to compute all solutions to an equation s=t, we apply narrowing steps to it until we obtain an equation s'=t' where s' and t' are unifiable. 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 (see [17] for a detailed survey). In the following we consider *normalizing innermost narrowing* derivations [10] where

- the narrowing step is performed at the leftmost innermost subterm, and
- the term is simplified to its normal form before a narrowing step is performed by applying rewrite rules from innermost to outermost positions.

The *innermost* strategy provides an efficient implementation [12, 13, 21, 23] while the *normalization* process is important since it prefers deterministic computations: rewriting a term to normal form can be done in a deterministic way since every rewrite sequence yields the same result (because $\mathcal{R}$ is convergent) 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 as 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 innermost narrowing is complete if $\mathcal{R}$ is convergent and all functions are totally defined, i.e., reducible on all appropriate constructor terms [10]. This is a reasonable class from the functional programming point of view. But it is also possible to extend this strategy to incompletely defined operations. In this case a so-called *innermost reflection* rule must be added which skips an innermost function call that cannot be evaluated [19]. For the sake of simplicity we assume in the following that all functions are totally defined, i.e., normalizing innermost narrowing is sufficient to compute all solutions.

3 Modes for functional logic programs

In pure logic programs, the *mode* for a predicate is a description of the possible arguments of a predicate when it is called [35]. E.g., the mode $p(\boldsymbol{g}, \boldsymbol{f}, \boldsymbol{a})$ specifies that the first argument is a ground term, the second argument is a free variable, and the third argument is an arbitrary term for all calls to predicate p. The mode information is useful to optimize the compiled code, i.e., to specialize the unification instructions and indexing scheme for a predicate [24, 25, 32, 34, 35]. Since functional logic languages are usually based on narrowing which uses unification to apply a function to a subterm, mode information could also be useful to optimize functional logic programs. However, the notion of “mode” in functional logic programs is different from pure logic programs if normalization is included in the narrowing process because functions are evaluated by narrowing as well as by rewriting. In the following we discuss this problem and define a new notion of modes for functional logic programs which will be used in Section 4 to optimize functional logic programs.

Example 2. In this example we discuss a derivation w.r.t. our narrowing strategy. Consider the rules of Example 1 together with the following rewrite rules:

$$
\begin{array}{rcll}
\texttt{double(0)} & \rightarrow & \texttt{0} & (R_3) \\
\texttt{double(s(N))} & \rightarrow & \texttt{s(s(double(N)))} & (R_4) \\
\texttt{quad(N)} & \rightarrow & \texttt{(N+N)+double(N)} & (R_5)
\end{array}
$$

We want to compute solutions to the initial equation `quad(X)=4` by our strategy, where 4 denotes the term `s(s(s(s(0))))`. Before applying any narrowing step, the equation is reduced to its normal form by rewrite steps. Hence we apply rule R_5 to the subterm `quad(X)`:

$$\texttt{quad(X)=4} \;\; \rightarrow_{\mathcal{R}} \;\; \texttt{(X+X)+double(X)=4}$$

Then the resulting equation is normalized by trying to apply rewrite rules to the three operation symbols, but no rewrite rule is applicable due to the free variable `X`. Hence the equation is already in normal form. Now a narrowing step is applied at the leftmost innermost position, i.e., the subterm `X+X`. Both rules R_1 and R_2 are applicable. We choose rule R_2 so that `X` is instantiated to `s(Y)`:

$$\texttt{(X+X)+double(X)=4} \;\; \leadsto_{\{\texttt{X}\mapsto\texttt{s(Y)}\}} \;\; \texttt{s(Y+s(Y))+double(s(Y))=4}$$

The resulting equation must be reduced to its normal form by trying to apply rewrite steps from innermost to outermost positions. A rewrite rule is not applicable to the leftmost innermost subterm `Y+s(Y)` since the first argument `Y` is a free variable. But we can apply rule R_4 to the subterm `double(s(Y))` and rule R_2 to the outer occurrence of `+`:

$$
\begin{array}{rll}
\texttt{s(Y+s(Y))+double(s(Y))=4} & \rightarrow_{\mathcal{R}} & \texttt{s(Y+s(Y))+s(s(double(Y)))=4} \\
 & \rightarrow_{\mathcal{R}} & \texttt{s((Y+s(Y))+s(s(double(Y))))=4}
\end{array}
$$

The latter equation is in normal form. Therefore we apply a narrowing step to the leftmost innermost subterm `Y+s(Y)`. We choose rule R_1 so that `Y` is instantiated to `0`:

$$\texttt{s((Y+s(Y))+s(s(double(Y))))=4} \;\leadsto_{\{\texttt{Y}\mapsto\texttt{0}\}}\; \texttt{s(s(0)+s(s(double(0))))=4}$$

We normalize the resulting equation by applying rule R_3 to `double(0)` and rules R_2 and R_1 to the remaining occurrence of `+`:

$$
\begin{array}{rll}
\texttt{s(s(0)+s(s(double(0))))=4} & \rightarrow_{\mathcal{R}} & \texttt{s(s(0)+s(s(0)))=4} \\
 & \rightarrow_{\mathcal{R}} & \texttt{s(s(0+s(s(0))))=4} \\
 & \rightarrow_{\mathcal{R}} & \texttt{s(s(s(s(0))))=4}
\end{array}
$$

Thus we have computed the solution $\{\texttt{X} \mapsto \texttt{s(0)}\}$ since the left- and right-hand side of the final equation are identical. A closer look to the narrowing and rewrite attempts in this derivation yields the following facts:

1. The operation `+` is evaluated both by narrowing and rewrite steps.

2. If a narrowing step is applied to `+`, the first argument is always free and the second argument may be partially instantiated.
3. If a rewrite step is applied to `+`, both arguments may be partially instantiated.
4. At the time when a narrowing step could be applied to `double` (i.e., if all functions to the left of `double` are evaluated), its argument is ground. Hence `double` is evaluated by rewriting and not by narrowing.
5. If a rewrite step is applied to `double`, its argument may be partially instantiated.
6. If a rewrite or narrowing step is applied to `quad`, its argument is always a free variable. Hence no rewrite rules can be applied to any function call in the right-hand side of rule R_5 immediately after the application of these rule, i.e., the rewrite attempts for these function calls can be skipped.

In order to have a formal representation of these properties, we assign to each operation a *narrowing mode* (`+(`$\boldsymbol{f}$`,`$\boldsymbol{a}$`)`, `double(`$\boldsymbol{g}$`)`, `quad(`$\boldsymbol{f}$`)` in this example) and a *rewrite mode* (`+(`$\boldsymbol{a}$`,`$\boldsymbol{a}$`)`, `double(`$\boldsymbol{a}$`)`, `quad(`$\boldsymbol{f}$`)`). Using this kind of mode information it is possible to avoid unnecessary rewrite attempts, compile rewrite derivations in a more efficient way, delete unnecessary rewrite or narrowing rules etc. (see Section 4). □

In the following we give a precise definition of the possible modes for functional logic programs w.r.t. a normalizing narrowing semantics. In this definition we consider a mode as a (possibly infinite) set of term tuples. Such a set contains all possible parameters which may occur in a function call. In subsequent sections we abstract such a set to a finite representation like $\boldsymbol{g}$, $\boldsymbol{f}$ or $\boldsymbol{a}$. Since there are also other useful abstractions (e.g., type approximations [4]), we do not restrict the general definition of modes.

Definition 2. Let f/n be an operation symbol and $N, R \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})^n$.
(a) N is called $\mathcal{N}$-mode (*narrowing mode*) for f/n whenever $\langle t_1, \ldots, t_n \rangle \in N$ if a narrowing step should be applied to the subterm $f(t_1, \ldots, t_n)$ during program execution.
(b) R is called $\mathcal{R}$-mode (*rewrite mode*) for f/n whenever $\langle t_1, \ldots, t_n \rangle \in R$ if a rewrite step should be applied to the subterm $f(t_1, \ldots, t_n)$ during program execution. □

We have defined modes w.r.t. arbitrary program executions. However, for the sake of good program optimizations it is desirable to consider only executions w.r.t. a given class of initial goals. In this case the modes are computed by a top-down analysis of the program starting from the initial goals.

4 Optimization of functional logic programs using modes

As mentioned in the previous section, we are not interested in the precise term sets contained in the modes, but we abstract these term sets into a finite number of *abstract values*. For the optimizations techniques we have in mind the abstract values $\boldsymbol{g}$, $\boldsymbol{f}$ and $\boldsymbol{a}$ are sufficient, where $\boldsymbol{g}$ denotes the set $\mathcal{T}(\mathcal{F}, \varnothing)$ of ground terms, $\boldsymbol{f}$ the set $\mathcal{X}$ of free variables and $\boldsymbol{a}$ the set $\mathcal{T}(\mathcal{F}, \mathcal{X})$ of all terms. Hence the $\mathcal{N}$-mode $\langle \boldsymbol{g}, \boldsymbol{a}, \boldsymbol{f} \rangle$ for the operation $f/3$ specifies that the first argument is ground and the third argument is a free variable if a narrowing rule should be applied to this operation. Such modes can be specified by the programmer, but it is more reliable to derive the modes automatically from the given program (w.r.t. a mode for the initial goal). Automatic mode inference has been investigated for pure logic programming (e.g., [3, 5, 6, 25, 30]) and similar schemes for functional logic programs are under development [18]. In the following we show possible optimization techniques w.r.t. given modes for a functional logic program.

4.1 Using freeness information

We have seen in Example 2 that rewrite steps cannot be applied to function calls if some arguments are not sufficiently instantiated. Hence we can omit all rewrite attempts to a function call if an argument that is required in all rewrite rules has $\mathcal{R}$-mode $\boldsymbol{f}$.

We say an operation f *requires argument* i if $t_i \notin \mathcal{X}$ for all rewrite rules $f(t_1, \ldots, t_n) \to r$, i.e., t_i has a constructor at the top. Our optimization w.r.t. freeness is based on the following proposition.

Proposition 3. *If an operation f has $\mathcal{R}$-mode $\langle m_1, \ldots, m_n \rangle$ with $m_i = \boldsymbol{f}$ and requires argument i, then no rewrite step can be applied to an f-rooted term during execution.*

In this case all rewrite rules for f can be deleted in the compiled program and all attempts to rewrite f-rooted subterms can be immediately skipped. However, in practice this case rarely occurs since rewrite steps are always applied to the entire goal before each single narrowing step. Therefore function arguments are usually not definitely free for all rewrite attempts but become more and more instantiated while narrowing steps are performed. But we can see in Example 2 that there is an interesting situation where unnecessary rewrite attempts occur. After applying a narrowing step with rule $l \to r$ to the leftmost innermost subterm, due to the eager normalization strategy, applications of rewrite rules are tried to all functions occurring in r. Since a narrowing step is only applied because of the insufficient instantiation of arguments (otherwise the subterm would be evaluated by rewriting), it is often the case that the function calls in r are not sufficiently instantiated to apply rewrite rules. Hence the rewrite attempts immediately after a narrowing step could be avoided.

In order to give a precise definition of this optimization, we define a special kind of rewrite mode which is valid immediately after a narrowing step.

Definition 4. Let $f(t_1, \ldots, t_n) \to r$ be a narrowing rule and N be a $\mathcal{N}$-mode for f/n. Let $g(s_1, \ldots, s_m)$ be a function call in r and $R_f \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})^m$. Then R_f is called $\mathcal{R}/\mathcal{N}$-mode (w.r.t. to N) (*rewrite mode w.r.t. narrowing*) for the function call $g(s_1, \ldots, s_m)$ iff $\sigma(\langle s_1, \ldots, s_n \rangle) \in R_f$ for each most general unifier σ of $\langle t_1, \ldots, t_n \rangle$ and some $\langle t'_1, \ldots, t'_n \rangle \in N$. □

Note that suitable $\mathcal{R}/\mathcal{N}$-modes can be easily derived from a given $\mathcal{N}$-mode of an operation. Since Proposition 3 is also valid w.r.t. $\mathcal{R}/\mathcal{N}$-modes and the immediate rewrite attempts after a narrowing step, we can use $\mathcal{R}/\mathcal{N}$-modes to avoid unnecessary rewrite attempts. For instance, consider Example 2 and the rule

```
s(M) + N → s(M + N)          (R2)
```

Since `+` has $\mathcal{N}$-mode $\langle \boldsymbol{f}, \boldsymbol{a} \rangle$, a suitable $\mathcal{R}/\mathcal{N}$-mode of the function call `M+N` in the right-hand side is $\langle \boldsymbol{f}, \boldsymbol{a} \rangle$. Therefore no rewrite rule is applicable to `M+N` immediately after a narrowing step with R_2 because `+` requires its first argument.

In the case of nested function calls, we can also skip rewrite attempts to function calls which contain function calls in normal form at a required argument position. For instance, if `(X+Y)+Z` occurs in the right-hand side of a narrowing rule and the $\mathcal{N}$-mode implies that `X` is always a free variable, then rewrite attempts to both occurrences of `+` can be neglected.

The realization of this optimization in a compiler-based implementation of normalizing innermost narrowing is easy. In order to avoid a dynamic search in the current goal for the leftmost innermost subterm, it is useful to manage an *occurrence stack* at run time [13]. This stack contains references to all functions calls in a goal in leftmost innermost order, i.e., the top element refers to the leftmost innermost subterm. If a narrowing rule $l \to r$ is applied, the top element of the occurrence stack is deleted, references to all function calls in r are added, and the application of rewrite rules are tried to all subterms referred by the occurrence stack.[5] The management of the occurrence stack provides an efficient implementation and causes nearly no overhead (see [13] for benchmarks). Moreover, it provides a simple realization of the freeness optimization. To skip unnecessary rewrite attempts in the right-hand side of a narrowing rule, the occurrences of the corresponding subterms are not pushed onto the occurrence stack. Although this optimization is simple, it has measurable effects on the execution time if the portion of narrowing steps in the computation is not too low (see Section 5 for benchmarks). In extreme cases all unnecessary rewrite attempts are avoided by this optimization.

4.2 Using groundness information

An implementation of normalizing narrowing requires the application of rewrite rules to all function calls in a goal before a narrowing step is performed. Therefore function calls cannot be represented by pieces of code similarly to predicate calls in the WAM [36], but they must be explicitly represented as a term structure. For instance, if the `quad` rule R_5 of Example 2 is applied in a narrowing or rewrite step, the term representation of the right-hand side `(N+N)+double(N)` is created in the heap area (which contains all term structures during program execution [13, 36].)[6] This implementation has the disadvantage that many terms are created on the heap which are garbage after the evaluation of the function calls. The situation can be improved if it is known that some functions are completely evaluable by rewriting. A sufficient criterion is the groundness of some arguments.[7]

[5] This explanation is slightly simplified. In the concrete implementation, a second so-called *copy occurrence stack* is used in the rewrite process. See [13] for more details.

[6] It is not necessary to create a term representation for all functions calls. Since the leftmost innermost function call `N+N` is evaluated in the next step, a representation of this term is only necessary if no rewrite rule is applicable to it. Therefore the creation of this term is delayed in [13]. This results in an implementation similar to WAM-based Prolog systems.

[7] Note that we assume that all narrowing rules are also used for rewriting, otherwise the proposition does not hold.

Proposition 5. *If an operation f has $\mathcal{R}$-mode $\langle \boldsymbol{g}, \ldots, \boldsymbol{g} \rangle$, then all f-rooted subterms are completely evaluated by rewriting during execution.*

This property holds since a narrowing step is only performed at an innermost position if some arguments are not sufficiently instantiated, but the latter condition can never be satisfied if it is a ground function call. Consequently, ground function calls can be implemented by a fixed sequence of function calls which do not require a representation on the heap. For instance, if `quad` has $\mathcal{R}$-mode $\langle \boldsymbol{g} \rangle$, then the rewrite rule `quad(N)`$\to$`(N+N)+double(N)` could be translated similarly to functions in imperative or functional languages according to the following code sequence:

```
N  := A1          % Register A1 contains the actual argument of quad
N1 := N+N         % call operation +
N2 := double(N)   % call operation double
N3 := N1+N2       % call operation +
return(N3)        % return the computed value
```

The intermediate values could be stored in an environment on the local stack which can be deleted after the `return` (or before, if last call optimization is implemented). Thus, if groundness information is available, we could optimize the code such that function calls need not be represented on the heap and intermediate results are stored on the local stack instead of the heap. This has the advantage that the used memory space on the local stack is automatically released after deterministic computations while the heap is cleaned up only after a garbage collection phase. Some results to this optimization are shown in Section 5.

4.3 Code elimination using mode information

Rewrite steps and narrowing steps differ in the application of the left-hand side to a subterm: while the subterm is *matched* with the left-hand side in a rewrite step, it is *unified* with the left-hand side in a narrowing step. Due to this different behavior (and some other reasons, cf. [13]), rewrite rules and narrowing rules are compiled into separate instructions. In particular, if the program rules defining operations are used both as narrowing rules and rewrite rules, each rule is compiled in two ways. This has a positive effect on the time efficiency of the compiled code, but it doubles the code space. On the other hand, only a few rules are actually used both for narrowing and rewriting in practical programs. Some rules are only used in rewrite steps, while others are exclusively used in narrowing steps. Information about modes can help to detect these cases at compile time so that unnecessary code can be avoided in the target program. The following conditions are sufficient criteria to omit rules in the target program:

1. If f has $\mathcal{R}$-mode $\langle m_1, \ldots, m_n \rangle$ with $m_i = \boldsymbol{f}$, then rewrite rules of the form $f(t_1, \ldots, t_n) \to r$ with $t_i \notin \mathcal{X}$ are superfluous (by Proposition 3).
2. Narrowing rule $f(t_1, \ldots, t_n) \to r$ is superfluous if f has $\mathcal{N}$-mode $\langle m_1, \ldots, m_n \rangle$ and for each $t_i \notin \mathcal{X}$ and each $t_i \in \mathcal{V}ar(t_j)$ (for some $j \neq i$) $m_i = \boldsymbol{g}$ holds (since in this case the rule is always applicable in a preceding rewrite step.)[8]

[8] Note that the case $t_i \in \mathcal{V}ar(t_j)$ is necessary since we allow multiple occurrences of the same variable in the left-hand side of a rule. E.g., the rule `f(X,X)`$\to$`X` is not applicable

Extreme cases of 2 are rules of the form $f(X_1, \ldots, X_n) \to r$ where $X_1, \ldots, X_n$ are pairwise different variables, or all narrowing rules for a function f which has $\mathcal{N}$-mode $\langle \boldsymbol{g}, \ldots, \boldsymbol{g} \rangle$.

For instance, in Example 2 we can delete R_3, R_4, R_5 as narrowing rules. These rules are only used in rewrite steps, while rules R_1 and R_2 are used both in rewrite and narrowing steps.

5 Experimental results

In order to obtain results about the practical usefulness of the optimizations discussed so far, we have applied these optimizations to some functional logic programs. These optimizations were performed with the ALF system [12, 13] which uses normalizing innermost narrowing as the operational semantics. We have not introduced any new low-level instructions into the abstract machine A-WAM on which the ALF system is based. All the optimizations discussed in Section 4 are implemented using the standard instruction set of the A-WAM which is the simplest, but not the most efficient way to implement these optimizations. Therefore it is obvious that better results can be obtained if the A-WAM would be redesigned according to the availability of mode information.

Table 1 shows the difference of the execution time between programs compiled without and with the optimizations w.r.t. freeness information as discussed in Section 4.1. All programs were executed on a Sparc 1. The programs are small but typical functional logic programs in the sense that functions are called with non-ground arguments so that narrowing rules must be applied to evaluate these functions. `arith` is a program that solves the equation `X+X=10` on natural numbers (where natural numbers are represented by terms built from the constructors `0` and `s`). `hamilton` computes a Hamiltonian path in a graph. `last` computes the last element of a given list with 10 elements by solving the equation `append(_,[E])=[···]`. `path` computes a complete path through a graph. `permsort` is the functional version of the permutation sort program, a typical generate-and-test program which demonstrates the advantages of functional logic programs compared to pure logic programs [14].

Program	Standard	Optimized	Improvement
`arith`	2.70	2.42	11.5%
`hamilton`	1180	980	20.4%
`last`	5.40	4.80	12.5%
`path`	1400	1120	25.0%
`permsort`	1680	1480	13.5%

Table 1. Execution times (in msec) for optimized programs w.r.t. freeness information

Although freeness information is only used to avoid some unnecessary rewrite attempts for the right-hand side after a narrowing step (and not for other more

to the term `f(Y,Z)` in a rewrite step, thus this rule must be kept as a narrowing rule.

	Standard		Optimized	
Program	local stack	heap	local stack	heap
`fac`	104	441168	161380	370104
`fib`	104	1145148	780	926248
`zero`	104	655620	636	280

Table 2. Maximum memory usage for optimized programs w.r.t. groundness information (in bytes)

primitive optimizations [24, 31, 32, 34, 35]), the table presents interesting improvements in the execution time. The variations show that it is difficult to state a general factor of improvement using freeness information. This factor largely depends on the number of function calls which can be safely skipped in the normalization process after the application of a narrowing rule.

Table 2 shows the memory usage for unoptimized and optimized programs w.r.t. groundness information as discussed in Section 4.2. The programs are recursive functions on natural numbers where natural numbers are represented by terms built from the constructors `0` and `s`. `fac` computes the factorial of 8, `fib` computes the 20'th Fibonacci number, and `zero` is a function which maps all inputs to the constant `0` but it is recursively defined similarly to `fib`.

Since we have not changed the instruction set of the A-WAM, we could only simulate the optimizations with the existing instruction set. But we can see in Table 2 that the heap space is reduced while the local stack increases. This is a desirable property since the local stack is automatically cleaned up after deterministic computations while the heap space must be reclaimed by a garbage collector. In the optimized version, no function calls are created on the heap. The remaining heap cells are occupied by constructor terms created during execution (in these examples: `s`-terms representing natural numbers). An extreme case is the recursive function `zero` which creates no constructor terms. The large heap space in the unoptimized version is due to the representation of recursive function calls in the heap.

6 Automatic derivation of modes

The main motivation of this paper is to show opportunities to optimize functional logic programs. For this purpose we have defined a notion of modes which is suitable for the particular operational semantics. However, the automatic derivation of these modes is another complex topic which will be addressed in a forthcoming paper [18]. In this section we will discuss some peculiarities related to the automatic derivation of modes.

Innermost narrowing without normalization is equivalent to SLD-resolution if the functional logic program is transformed into a flat program without nested function calls [2]. For instance, we could transform the rules of Example 1 into the flat logic program

```
add(0,N,N).
add(s(M),N,s(Z)) :- add(M,N,Z).
```

where the predicate `add` corresponds to the function `+` with its result value. The nested function call in the right-hand side of rule R_2 has been replaced by the new variable `Z` and the additional condition `add(M,N,Z)`. Now each innermost narrowing derivation w.r.t. rules R_1 and R_2 corresponds to one SLD-derivation w.r.t. the transformed logic program.

Due to these similarities of narrowing and SLD-resolution, one could try to apply abstract interpretation techniques developed for logic programming (e.g., [3, 22, 27]) to derive the desired information. E.g., to derive the narrowing mode of the function `+` w.r.t. to the class of initial goals x+y=z, where x and y are always ground and z is a free variable, we could use an abstract interpretation framework for logic programming to infer the call modes of the predicate `add` w.r.t. the class of initial goals `add(`x`,`y`,`z`)`. In this case we infer that the call mode is $\langle \boldsymbol{g}, \boldsymbol{g}, \boldsymbol{f} \rangle$ and the argument z of the initial goal will be bound to a ground term at the end of a successful computation. Hence we could deduce that $\langle \boldsymbol{g}, \boldsymbol{g} \rangle$ is the narrowing mode of the function `+`.

However, normalizing narrowing, which we have considered in this paper, does not directly correspond to SLD-resolution because of the intermediate normalization process. These normalization steps between narrowing steps may delete entire subterms or change the order of subterms. These subtleties require more sophisticated analysis techniques than those developed for pure logic programming. E.g., consider the rules

```
f(0,Z) → 0                                g(0) → 0
```

and the initial equation `f(g(X),g(Y))=0`. Using normalizing innermost narrowing, this equation is solved by applying a narrowing step to the innermost subterm `g(X)` followed by a rewrite step:

$$\begin{array}{rll} \texttt{f(g(X),g(Y)) = 0} & \leadsto_{\{\texttt{X}\mapsto\texttt{0}\}} & \texttt{f(0,g(Y)) = 0} \\ & \to_{\mathcal{R}} & \texttt{0 = 0} \end{array}$$

Hence variable `Y` remains unbound at the end of the computation. On the other hand, the flattening transformation yields the following corresponding logic program:

```
f(0,Z,0).
g(0,0).
?- g(X,Z1), g(Y,Z2), f(Z1,Z2,0).
```

But this logic program has another behavior than the functional logic program since the variable `Y` will be bound by SLD-resolution! Therefore we can apply abstract interpretation frameworks for logic programming in our context only if there are no rewrite rules which may delete or permute arguments. Such rewrite rules require a special treatment in the abstract interpretation procedure which will be described in a forthcoming paper [18]. Another approach to abstract interpretation of functional logic programs based on an alternative operational semantics is described in [16].

7 Conclusions

In this paper we have shown optimization techniques in the presence of mode information which are unique to the execution mechanism of functional logic programs. We have considered normalizing innermost narrowing as the operational semantics since it has been shown that this strategy is a reasonable improvement over Prolog's left-to-right resolution strategy [10, 14]. We have defined the notion of modes for functional logic programs. These modes can be used to optimize the normalization process. On the one hand, the normalization process is the reason for the operational improvements of functional logic languages compared to pure logic languages. On the other hand, the normalization process may add unnecessary work. This can be improved using modes: freeness information avoids superfluous rewrite attempts, and groundness information provides for a better implementation (in terms of memory consumption) of the normalization process. Moreover, information about modes can also be used to avoid the generation of code for rewrite or narrowing rules which will never be used at run time.

Future work includes a refinement of the abstract machine for the execution of functional logic programs following the lines presented in [32, 34], the development of appropriate abstract interpretation frameworks to derive mode information at compile time [18], and refined applicability conditions for rewrite rules using type information [4].

References

1. D. Bert and R. Echahed. Design and Implementation of a Generic, Logic and Functional Programming Language. In *Proc. ESOP'86*, pp. 119–132. Springer LNCS 213, 1986.
2. P.G. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-Resolution. *Theoretical Computer Science 59*, pp. 3–23, 1988.
3. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming (10)*, pp. 91–124, 1991.
4. M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inferencing. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 669–683, 1988.
5. S.K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM TOPLAS*, Vol. 11, No. 3, pp. 418–450, 1989.
6. S.K. Debray and D.S. Warren. Automatic Mode Inference for Logic Programs. *Journal of Logic Programming (5)*, pp. 207–229, 1988.
7. D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations, and Equations*. Prentice Hall, 1986.
8. 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.
9. 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.
10. 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.
11. 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.
12. M. Hanus. Compiling Logic Programs with Equality. In *Proc. PLILP'90*, pp. 387–401. Springer LNCS 456, 1990.
13. M. Hanus. Efficient Implementation of Narrowing and Rewriting. In *Proc. PDK'91*, pp. 344–365. Springer LNAI 567, 1991.

14. M. Hanus. Improving Control of Logic Programs by Using Functional Logic Languages. In *Proc. PLILP'92*, pp. 1–23. Springer LNCS 631, 1992.
15. M. Hanus. Incremental Rewriting in Narrowing Derivations. In *Proc. ALP'92*, pp. 228–243. Springer LNCS 632, 1992.
16. M. Hanus. On the Completeness of Residuation. In *Proc. of the 1992 Joint Int. Conf. and Symp. on Logic Programming*, pp. 192–206. MIT Press, 1992.
17. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *To appear in Journal of Logic Programming*, 1994.
18. M. Hanus and F. Zartmann. Automatic derivation of modes for functional logic programs. Max-Planck-Institut für Informatik, Saarbrücken (in preparation), 1994.
19. S. Hölldobler. *Foundations of Equational Logic Programming*. Springer LNCS 353, 1989.
20. J.-M. Hullot. Canonical Forms and Unification. In *Proc. 5th Conference on Automated Deduction*, pp. 318–334. Springer LNCS 87, 1980.
21. H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Graph-based Implementation of a Functional Logic Language. In *Proc. ESOP'90*, pp. 271–290. Springer LNCS 432, 1990.
22. B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A Generic Abstract Interpretation Algorithm and its Complexity Analysis. In *Proc. International Conference on Logic Programming*, pp. 64–78. MIT Press, 1991.
23. R. Loogen. Relating the Implementation Techniques of Functional and Functional Logic Languages. *New Generation Computing*, Vol. 11, pp. 179–215, 1993.
24. A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The impact of abstract interpretation: an experiment in code generation. In *Proc. Sixth International Conference on Logic Programming (Lisboa)*, pp. 33–47. MIT Press, 1989.
25. C.S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming (1)*, pp. 43–66, 1985.
26. 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.
27. U. Nilsson. Systematic Semantic Approximations of Logic Programs. In *Proc. PLILP'90*, pp. 293–306. Springer LNCS 456, 1990.
28. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Int. Symp. on Logic Programming*, pp. 138–151, Boston, 1985.
29. 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.
30. Z. Somogyi. A system of precise modes for logic programs. In *Proc. Fourth Int. Conf. on Logic Programming*, pp. 769–787. MIT Press, 1987.
31. A. Taylor. Removal of Dereferencing and Trailing in Prolog Compilation. In *Proc. Sixth Int. Conf. on Logic Programming*, pp. 48–60. MIT Press, 1989.
32. A. Taylor. LIPS on a MIPS: Results form a Prolog Compiler for a RISC. In *Proc. Seventh Int. Conf. on Logic Programming*, pp. 174–185. MIT Press, 1990.
33. P. Van Roy. An Intermediate Language to Support Prolog's Unification. In *Proc. 1989 North American Conf. on Logic Programming*, pp. 1148–1164. MIT Press, 1989.
34. P.L. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Univ. of California Berkeley, 1990. Report No. UCB/CSD 90/600.
35. D.H.D. Warren. Implementing PROLOG - Compiling Logic Programs. 1 and 2. D.A.I. Research Report No. 39 and 40, University of Edinburgh, 1977.
36. D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Stanford, 1983.