In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation* (LOPSTR'95), Utrecht, Springer LNCS 1048, pp. 252–266, 1995

Efficient Translation of Lazy Functional Logic Programs into Prolog

Michael Hanus

Informatik II, RWTH Aachen
D-52056 Aachen, Germany
hanus@informatik.rwth-aachen.de

Abstract. In this paper, we present a high-level implementation of lazy functional logic programs by transforming them into Prolog programs. The transformation is controlled by generalized definitional trees which specify the narrowing strategy to be implemented. Since we consider a sophisticated narrowing strategy, a direct mapping of functions into predicates is not possible. Therefore, we present new techniques to reduce the interpretational overhead of the generated Prolog code. This leads to a portable and efficient implementation of functional logic programs.

1 Introduction

In recent years, a lot of proposals have been made to amalgamate functional and logic programming languages [15]. Functional logic languages with a sound and complete operational semantics are based on *narrowing*, a combination of the reduction principle of functional languages and the resolution principle of logic languages. Narrowing, originally introduced in automated theorem proving [26], is used to *solve* equations by finding appropriate values for variables occurring in arguments of functions. A *narrowing step* instantiates some variables in a goal and applies a reduction step to a redex of the instantiated goal. The instantiation of goal variables is usually computed by unifying a subterm of the goal with the left-hand side of some rule.

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

$$\begin{array}{rcll} 0+y & \to & y & (R_1) \\ s(x)+y & \to & s(x+y) & (R_2) \end{array} \qquad \begin{array}{lcll} 0 \leq x & \to & true & (R_3) \\ s(x) \leq 0 & \to & false & (R_4) \\ s(x) \leq s(y) & \to & x \leq y & (R_5) \end{array}$$

The equation $x+y \leq 0 \approx true$ can be solved by a narrowing step with rule R_1 followed by a narrowing step with rule R_3 so that x and y are instantiated to 0 and the instantiated equation is reduced to the trivial equation $true \approx true$:

$$x+y \leq 0 \approx true \leadsto_{\{x\mapsto 0\}} y \leq 0 \approx true \leadsto_{\{y\mapsto 0\}} true \approx true$$

Hence we have found the solution $\{x \mapsto 0, y \mapsto 0\}$ to the given equation. □

Similarly to functional languages, we have to fix the selection of positions for the next narrowing step in order to reduce the search space. Eager functional logic languages like ALF [12], eager-BABEL [18], or SLOG [8] apply narrowing steps at innermost positions. To ensure completeness, they require a terminating set of rewrite rules which prohibit the application of typical functional programming techniques like infinite data structures. Therefore, we are interested in lazy narrowing strategies [22, 25] where narrowing steps are applied at outermost positions in general and at an inner position only if it is demanded and contributes to some later narrowing step at an outer position. Although such a lazy strategy can avoid useless computation steps, it has been shown that this is not generally true if one does not take care of a controlled instantiation of logical variables [4]. However, for the class of inductively sequential programs, which covers typical functional programs, there is a strategy, called *needed narrowing* [4], which is optimal w.r.t. the length of the narrowing derivations and the number of computed solutions. Inductively sequential programs do not allow overlapping left-hand sides of the rewrite rules. However, in some applications, particularly in logic programming, such overlapping rules are useful. Unfortunately, overlapping rules may lead to nonterminating computations w.r.t. lazy narrowing strategies [11]. This can be avoided if lazy narrowing is combined with simplification between narrowing steps [14]. Therefore, we obtain a good lazy narrowing strategy if we apply needed narrowing on inductively sequential programs and integrate simplification for the remaining programs.

In this paper, we consider the high-level implementation of such a sophisticated narrowing strategy. To avoid a complex direct implementation based on a new abstract machine (see [15] for a survey on these implementation techniques), we follow the proposals presented in [2, 6, 17, 19]. We translate lazy functional logic programs into Prolog programs and obtain by this simple transformation a portable and efficient implementation of our narrowing strategy. The translation of eager narrowing strategies into Prolog is straightforward by flattening nested function calls [5]. However, the translation of lazy narrowing strategies is a challenging task, in particular, if narrowing is interleaved with simplification. Our solution is the first Prolog implementation of a lazy narrowing strategy which comprises simplification. Nevertheless, we obtain a better run-time behavior w.r.t. previous work since we apply partial evaluation techniques to the translated program.

In the next section, we recall basic notions and introduce our narrowing strategy. In Section 3, we present the translation of inductively sequential programs, whereas Section 4 contains the translation of arbitrary functional logic programs. Optimizations obtained by partial evaluation and the implementation of sharing are discussed in Sections 5 and 6, respectively. Finally, we discuss the efficiency of our translation techniques by means of some benchmarks.

2 Lazy Narrowing Strategies

We assume familiarity with basic notions of term rewriting [7]. We consider a *many-sorted signature* partitioned into a set $\mathcal{C}$ of constructors and a set $\mathcal{F}$ of functions. We write $c/n \in \mathcal{C}$ and $f/n \in \mathcal{F}$ for n-ary constructor and function symbols, respectively. The set of *terms* and *constructor terms* with variables from

$\mathcal{X}$ are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$. $\mathcal{V}ar(t)$ denotes the set of variables occurring in a term t. A *pattern* is a term of the form $f(t_1, \ldots, t_n)$ where $f/n \in \mathcal{F}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. A *head normal form* is a variable or a term of the form $c(t_1, \ldots, t_n)$ with $c/n \in \mathcal{C}$. A *position* p in a term t is represented by a sequence of natural numbers, $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s (see [7] for details).

A *term rewriting system* $\mathcal{R}$ is a set of *rewrite rules* $l \to r$ where l is a pattern and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. l and r are called left-hand side and right-hand side, respectively.[1] A rewrite rule is called a *variant* of another rule if it is obtained by a unique replacement of variables by other variables.

Narrowing is a method to compute solutions to an equation $s \approx t$. $t \leadsto_\sigma t'$ is a narrowing step if there are a nonvariable position p in t (i.e., $t|_p \notin \mathcal{X}$), a variant $l \to r$ of a rewrite rule of $\mathcal{R}$ with $\mathcal{V}ar(t) \cap \mathcal{V}ar(l) = \emptyset$, and a unifier[2] σ of $t|_p$ and l with $t' = \sigma(t[r]_p)$.[3] Since narrowing applies rewrite rules only in one direction, additional restrictions are necessary for the completeness of narrowing, i.e., we require the confluence of $\mathcal{R}$. This can be ensured by the following condition: if $l_1 \to r_1$ and $l_2 \to r_2$ are variants of rewrite rules and σ is a unifier for l_1 and l_2, then $\sigma(r_1) = \sigma(r_2)$ (*weak orthogonality*).

Since we do not require terminating term rewriting systems, normal forms may not exist. Therefore, we define the validity of an equation as a *strict equality* on terms [10, 22] by the following rules, where $\wedge$ is assumed to be a right-associative infix symbol.

$$
\begin{array}{cl}
c \approx c \to \mathit{true} & \forall c/0 \in \mathcal{C} \\
c(x_1, \ldots, x_n) \approx c(y_1, \ldots, y_n) \to (x_1 \approx y_1) \wedge \cdots \wedge (x_n \approx y_n) & \forall c/n \in \mathcal{C} \\
\mathit{true} \wedge x \to x &
\end{array}
$$

A solution of an *equation* $t_1 \approx t_2$ is computed by narrowing it to *true* with these rules. Since this simple narrowing procedure (enumerating all narrowing derivations) is very inefficient, several authors have proposed restrictions on the admissible narrowing derivations (see [15] for a detailed survey). We are interested in *lazy narrowing* [21, 25] which is influenced by the idea of lazy evaluation in functional programming languages. Lazy narrowing steps are only applied at outermost positions with the exception that arguments are evaluated by narrowing to their head normal form if their values are required for an outermost narrowing step. Since the notion of "required arguments" depends on the rule to be applied

[1] In this paper, we consider only unconditional rewrite rules for the sake of simplicity. Nevertheless, the presented implementation techniques can be extended to conditional rules (e.g., as done in [19]) and completeness results for the conditional case can be found in [16].

[2] In most papers, narrowing is defined with most general unifiers. As shown in [4], an optimal narrowing strategy which avoids superfluous steps can only be obtained if the restriction to mgu's is dropped. Therefore, we consider arbitrary unifiers. However, only a small subset of these unifiers are computed by our narrowing strategy.

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

and leaves some freedom, different lazy narrowing strategies have been proposed [4, 17, 19, 21, 22]. We will specify our narrowing strategy by the use of definitional trees, a concept introduced by Antoy [3] to define efficient normalization strategies.

$\mathcal{T}$ is called *generalized definitional tree* with pattern π iff one of the following cases holds:

$\mathcal{T} = rule(\pi \to r)$, where $\pi \to r$ is a variant of a rule in $\mathcal{R}$.

$\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, where π is a pattern, o is an occurrence of a variable in π, $c_1, \ldots, c_k$ are different constructors of the sort of $\pi|_o$ $(k > 0)$, and, for $i = 1, \ldots, k$, $\mathcal{T}_i$ is a generalized definitional tree with pattern $\pi[c_i(x_1, \ldots, x_n)]_o$, where n is the arity of c_i and $x_1, \ldots, x_n$ are new distinct variables.

$\mathcal{T} = or(\mathcal{T}_1, \ldots, \mathcal{T}_k)$, where $\mathcal{T}_1, \ldots, \mathcal{T}_k$ are generalized definitional trees with pattern π.

A *generalized definitional tree* of an n-ary function f is a generalized definitional tree $\mathcal{T}$ with pattern $f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are distinct variables, such that for each rule $l \to r$ with $l = f(t_1, \ldots, t_n)$ there is a node $rule(l' \to r')$ in $\mathcal{T}$ with l variant of l'. A *definitional tree* is a generalized definitional tree without *or*-nodes.[4] For instance, the definitional tree of the function $\leq$ in Example 1 is

$$\begin{aligned} branch(x \leq y, 1, &rule(\mathbf{0} \leq y \to true),\\ &branch(\boldsymbol{s}(x_1) \leq y, 2, rule(s(x_1) \leq \mathbf{0} \to false),\\ &\qquad rule(s(x_1) \leq \boldsymbol{s}(y_1) \to x_1 \leq y_1))) \end{aligned}$$

A function f is called *inductively sequential* if there exists a definitional tree of f such that each *rule* node corresponds to exactly one rule of $\mathcal{R}$. We denote this property by $f/n \in IS(\mathcal{R})$. The term rewriting system $\mathcal{R}$ is called inductively sequential if each function defined by $\mathcal{R}$ is inductively sequential.

A generalized definitional tree defines a strategy to apply narrowing steps.[5] To narrow a term t, we consider the generalized definitional tree $\mathcal{T}$ of the outermost function symbol of t (note that, by definition of strict equality, the outermost symbol is always a function if we narrow equations):

$\mathcal{T} = rule(\pi \to r)$**:** Apply rule $\pi \to r$ to t (note that t is always an instance of π).

$\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$**:** Consider the subterm $t|_o$.

1. If $t|_o$ has a function symbol at the top, we narrow this subterm (to a head normal form) by recursively applying our strategy to $t|_o$.
2. If $t|_o$ has a constructor symbol at the top, we narrow t with $\mathcal{T}_j$, where the pattern of $\mathcal{T}_j$ unifies with t, otherwise (if no pattern unifies) we fail.
3. If $t|_o$ is a variable, we nondeterministically select a subtree $\mathcal{T}_j$, unify t with the pattern of $\mathcal{T}_j$ (i.e., $t|_o$ is instantiated to the constructor of the pattern of $\mathcal{T}_j$ at position o), and narrow this instance of t with $\mathcal{T}_j$.

$\mathcal{T} = or(\mathcal{T}_1, \ldots, \mathcal{T}_k)$**:** Nondeterministically select a subtree $\mathcal{T}_j$ and proceed narrowing t with $\mathcal{T}_j$.

[4] This corresponds to Antoy's notion [3] except that we ignore *exempt* nodes.

[5] Due to lack of space, we omit a precise definition which can be found in [4] for inductively sequential systems and in [19] for generalized definitional trees.

For definitional trees (i.e., without *or* nodes), this strategy is called *needed narrowing* [4] which is the currently best narrowing strategy due to its optimality w.r.t. the length of derivations (if terms are shared, compare Section 6) and the number of computed solutions. For instance, the rewrite system of Example 1 is inductively sequential and the successful derivation is a needed narrowing derivation. There is only one further needed narrowing derivation for this goal, which is not successful:

$$x + y \leq 0 \approx true \ \leadsto_{\{x \mapsto s(x_1)\}} \ s(x_1 + y) \leq 0 \approx true \ \leadsto_{\{\}} \ false \approx true$$

Note that the equivalent Prolog program obtained by flattening [5] has an infinite search space, since the first literal of the goal "`add(X,Y,Z),leq(Z,0,true)`" has infinitely many solutions (which can be avoided by additional delay declarations [23]; however, this may cause the loss of completeness).

We consider generalized definitional trees as a part of the program since they specify the concrete evaluation strategy (like when/wait declarations in Prolog systems). However, the user can also omit the trees since there are various methods to construct them (e.g., [19]).

3 Translation of Inductively Sequential Programs

In this section, we assume that $\mathcal{R}$ is inductively sequential. For this class of programs, it is shown in [4] that needed narrowing, i.e., narrowing with definitional trees, is an optimal strategy. To implement this strategy, we define three kinds of predicates in Prolog:

1. `A === B` is satisfied if `A` and `B` are strictly equal, i.e., `A` and `B` are reducible to a same ground constructor term. This predicate is implemented by repeated narrowing of `A` and `B` to head normal forms and comparing the outermost constructors (note that lazy narrowing reduces terms to head normal form and not to normal form).
2. `hnf(T,H)` is satisfied if `H` is a head normal form of `T`. If `T` is not in head normal form, `T` is narrowed using the strategy described above.
3. f_p`(`$t_1,\ldots,t_n$`,H)` is satisfied if `H` is a head normal form of $f(t_1,\ldots,t_n)$, where the subterms of $f(t_1,\ldots,t_n)$ at the positions in the set p are already in head normal form.

The clauses to define strict equality are straightforward:

$$\texttt{A === B :- hnf(A,HA), hnf(B,HB), seq(HA,HB).}$$
$$\texttt{seq(}c\texttt{(X}_1,\ldots,\texttt{X}_n\texttt{),}c\texttt{(Y}_1,\ldots,\texttt{Y}_n\texttt{)) :- X}_1\texttt{===Y}_1\texttt{,}\ldots\texttt{,X}_n\texttt{===Y}_n\texttt{.} \quad \forall c/n \in \mathcal{C}$$

The clauses to define `hnf` are also a straightforward translation of the definition of head normal form:

$$\texttt{hnf(T,T) :- var(T), !.}$$
$$\texttt{hnf(}f\texttt{(X}_1,\ldots,\texttt{X}_n\texttt{),H) :- !, }f_\emptyset\texttt{(X}_1,\ldots,\texttt{X}_n\texttt{,H).} \quad \forall f/n \in \mathcal{F}$$
$$\texttt{hnf(T,T).} \quad \texttt{\% T } \textit{is constructor-headed due to the previous clauses.}$$

The definition of the clauses for the predicates f_p`(X`$_1,\ldots,$`X`$_n$`,H)` is slightly more complicated but also an obvious translation of our previously described strategy.

We specify the generation of these clauses by a translation function *trans* which takes a definitional tree $\mathcal{T}$ with pattern π and a set p of already evaluated positions of π as input and yields a set of Prolog clauses. Each function f is translated by $trans(\mathcal{T},\emptyset)$ if $\mathcal{T}$ is a definitional tree of f.

$trans(rule(f(t_1,\ldots,t_n) \to r),p) \quad :=$

$\boxed{f_p\texttt{(}t_1,\ldots,t_n\texttt{,H) :- hnf(}r\texttt{,H).}}$

$trans(branch(\pi,o,\mathcal{T}_1,\ldots,\mathcal{T}_k),p) \quad :=$

$\boxed{f_p\texttt{(}t_1,\ldots,t_n\texttt{,H) :- hnf(}x\texttt{,Y), }f_{p\cup\{o\}}\texttt{(}t'_1,\ldots,t'_n\texttt{,H).}}$

$trans(\mathcal{T}_1,p\cup\{o\})$

$\ldots$

$trans(\mathcal{T}_k,p\cup\{o\})$

where $\pi = f(t_1,\ldots,t_n),\ \pi|_o = x,\ \pi[\mathtt{Y}]_o = f(t'_1,\ldots,t'_n)$

In these and all subsequent translation schemes, all unspecified variables occurring in the rules are new (here: `H` and `Y` are new variables). It is obvious that this translation scheme implements the narrowing strategy described above. To distinguish the different predicates corresponding to different nodes of $\mathcal{T}$, the predicate names are indexed by p. A *rule* node is translated into a clause which applies this rule by computing the head normal form of the right-hand side. For a *branch* node, the requested subterm is evaluated to head normal form followed by a call to the predicate corresponding to the immediate subtrees.

If we translate all rules of Example 1 by this scheme (the generated clauses are shown in Appendix A), we can compute solutions to the equation $z + s(0) \approx s(s(0))$ by proving the Prolog goal "`?- Z+s(0)===s(s(0)).`"

4 Translation of Lazy Narrowing with Simplification

Inductively sequential systems do not allow *or* nodes in the definitional trees, in particular, overlapping rules are not permitted. Nevertheless, overlapping rules sometimes occur in programs written in a logic programming style. Therefore, we consider in this section a term rewriting system $\mathcal{R}$ which may not be inductively sequential. Our translation scheme could be simply extended to such programs by defining the following additional rule to translate *or* nodes:

$trans(or(\mathcal{T}_1,\ldots,\mathcal{T}_k),p) \quad :=$

$trans(\mathcal{T}_1,p) \ \cdots \ trans(\mathcal{T}_k,p)$

This means that the different alternatives represented by an *or* node are translated into alternative clauses (this is identical to the translation scheme in [19]), and we obtain the behavior of (simple) lazy narrowing [21, 22, 25]. However, in the presence of overlapping rules, simple lazy narrowing has a high risk to run into infinite loops by selecting the "wrong" rule and evaluating the "wrong" argument to head normal form.

Example 2. Consider the following rules defining arithmetic operations:

$$\begin{array}{lllllll} 0 * x & \to & 0 & (R_1) & \quad one(0) & \to & s(0) & (R_3) \\ x * 0 & \to & 0 & (R_2) & \quad one(s(x)) & \to & one(x) & (R_4) \end{array}$$

To compute a solution to the equation $one(x) * 0 \approx 0$, we could choose rule R_1 to evaluate the left-hand side. Rule R_1 demands the evaluation of $one(x)$ to a head normal form. Unfortunately, there are infinitely many possibilities to evaluate $one(x)$, in particular, there is an infinite derivation using R_4 in each step:

$$one(x) * 0 \approx 0 \ \leadsto_{\{x \mapsto s(x_1)\}} \ one(x_1) * 0 \approx 0 \ \leadsto_{\{x_1 \mapsto s(x_2)\}} \ \cdots$$

This infinite loop can be avoided if the goal is *simplified* before a narrowing step is performed. Simplification is similar to narrowing but does not instantiate goal variables and is, therefore, a deterministic evaluation process. Since the term $one(x) * 0$ can be simplified to 0 by rule R_2, *lazy narrowing with simplification* [14] has a finite search space in this example. □

Lazy narrowing with simplification reduces the search space and is sound and complete if the set of rules used for simplification is terminating [14]. Moreover, simplification must be performed with the same strategy as narrowing (of course, without instantiating goal variables). Thus, we can define a similar translation scheme for simplification and call the predicates performing simplification before each narrowing step. However, simplification has no effect for inductively sequential systems due to the optimality of needed narrowing (see [14] for more details). Therefore, simplification should be applied only if a function $f/n \notin IS(\mathcal{R})$ occurs at run time. This leads to the following implementation scheme:

1. We generate the narrowing scheme of Section 3 for inductively sequential functions.
2. We generate a simplification scheme similar to the narrowing scheme. However, there are some important differences since simplification always succeeds and returns a simplified term which is not necessarily in head normal form.

The clauses of the predicate `hnf` are defined by the following modified scheme:

```
hnf(T,T) :- var(T), !.
hnf(f(X1,...,Xn),H) :- !, f∅(X1,...,Xn,H).    ∀f/n ∈ IS(R)
hnf(f(X1,...,Xn),H) :- !, simp(f(X1,...,Xn),T),
                          nstep(T,R,_), hnf(R,H).  ∀f/n ∉ IS(R)
hnf(T,T).
```

`simp` simplifies a term using the same strategy as narrowing, and `nstep` performs a single narrowing step on the simplified term. Due to the similarity of the strategies for simplification and narrowing, we implement simplification by a scheme similar to narrowing presented above. Thus, the predicate `simp` corresponds to the predicate `hnf` but with the difference that `simp` does not fail and always returns a simplified term (which may not be in head normal form if simplication rules are not applicable due to the insufficient instantiation of variables).

```
simp(T,T) :- var(T), !.
simp(f(X1,...,Xn),T) :- !, simp_{f,∅}(X1,...,Xn,T).   ∀f/n ∈ F
simp(T,T).
```

`simp` is called if a term `T` should be reduced to head normal form in order to apply a simplification step. The following translation scheme is similar to *trans*. It generates for each generalized definitional tree of a function f the clauses for simplifying a function call $f(\cdots)$:

$simptrans(rule(f(t_1,\ldots,t_n) \to r), p)$ $:=$

$\boxed{\texttt{simp}_{f,p}\texttt{(}t_1,\ldots,t_n\texttt{,R) :- !, simp(}r\texttt{,R).}}$

$simptrans(branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), p)$ $:=$

```
simp_{f,p}(t_1,...,t_n,R) :- !, simp(x,Y),
    (nonvar(Y) -> simp_{f,p∪{o}}(t'_1,...,t'_n,R) ; R=f(t'_1,...,t'_n) ).
```

$simptrans(\mathcal{T}_1, p \cup \{o\})$

...

$simptrans(\mathcal{T}_k, p \cup \{o\})$

$\boxed{\texttt{simp}_{f,p\cup\{o\}}\texttt{(}t_1,\ldots,t_n\texttt{,}f\texttt{(}t_1,\ldots,t_n\texttt{)).}}$

where $\pi = f(t_1,\ldots,t_n)$, $\pi|_o = x$, $\pi[\mathtt{Y}]_o = f(t'_1,\ldots,t'_n)$

The cuts in the generated rules emphasize the deterministic behavior of the simplification process. The final clause generated for each *branch* node is necessary to return the current term instead of causing a failure if no simplification rule is applicable. The condition `nonvar(Y)` in the translation of *branch* nodes is necessary to ensure that the goal variable `Y` is not instantiated in subsequent simplification rules (recall that this is the basic difference between simplification and narrowing). If `Y` is an unbound variable, then no simplification rules of the subtrees $\mathcal{T}_1,\ldots,\mathcal{T}_k$ are applicable. Hence, the simplified term $f(t'_1,\ldots,t'_n)$ is returned instead of applying further simplification rules.

Additionally, a node $or(\mathcal{T}_1,\ldots,\mathcal{T}_k)$ is processed by *simptrans*[6] by translating each $\mathcal{T}_j$ into separate Prolog predicates. However, the translation scheme for $\mathcal{T}_j$ is slightly changed for $j = 1,\ldots,k-1$. Instead of constructing the term $f(t_1,\ldots,t_n)$ if no rule is applicable, the simplification predicates corresponding to the generalized definitional tree $\mathcal{T}_{j+1}$ are called since $\mathcal{T}_{j+1}$ may contain alternative simplification rules (see Appendix B for the translation of the overlapping `*`-rules of Example 2).

The predicate `nstep` is responsible to perform a *single* narrowing step. For this purpose, an additional argument `C` is used which is instantiated iff a narrowing step has been applied. Therefore, we generate the clauses

```
nstep(T,T,C) :- var(T), !.
nstep(f(X_1,...,X_n),T,C) :- !, f_step_∅(X_1,...,X_n,T,C).   ∀f/n ∈ F
nstep(T,T,C).  % T is constructor-headed due to the previous clauses.
```

and clauses for each generalized definitional tree by the following scheme, which is a slightly modified translation scheme for narrowing rules:

$steptrans(rule(f(t_1,\ldots,t_n) \to r), p)$ $:=$

$\boxed{f\texttt{\_step}_p\texttt{(}t_1,\ldots,t_n\texttt{,}r\texttt{,step).}}$ % *instantiate control variable to* `step`

$steptrans(branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), p)$ $:=$

```
f_step_p(t_1,...,t_n,R,C) :- nstep(x,Y,C),
    (var(C) -> f_step_{p∪{o}}(t'_1,...,t'_n,R,C) ; R=f(t'_1,...,t'_n) ).
```

$steptrans(\mathcal{T}_1, p \cup \{o\})$

...

[6] Due to space limitations, we do not show the formal definition.

$steptrans(\mathcal{T}_k, p \cup \{o\})$
where $\pi = f(t_1, \ldots, t_n),\ \pi|_o = x,\ \pi[\mathtt{Y}]_o = f(t_1', \ldots, t_n')$

$steptrans(or(\mathcal{T}_1, \ldots, \mathcal{T}_k), p) \quad :=$
$steptrans(\mathcal{T}_1, p)$
$\ldots$
$steptrans(\mathcal{T}_k, p)$

Due to the condition `var(C)->`$\cdots$ in clauses corresponding to branch nodes, the predicate f`_step`$_p$ may not return a head normal form but performs only one narrowing step. All clauses generated by our scheme for Example 2 are shown in Appendix B. The size of the translated programs is approximately doubled in comparison to the translation without the simplification scheme. This is due to the fact that each rule can be applied in a "narrowing mode" and a "simplification mode" which requires different implementations.

Since the rewrite rules are separately translated into clauses for narrowing and simplification, we can also choose different rewrite rules for narrowing and simplification. Actually, the programmer has to specify a terminating subset of $\mathcal{R}$ which is used for simplification in order to ensure completeness (see [14]). Moreover, it has been argued in [8] that it is sensible to use additionally inductive consequences or CWA-valid rules for simplification. All this is supported by our separate translation of narrowing and simplification rules.

5 Optimization by Partial Evaluation

It is not surprising that our general translation scheme contains many opportunities for optimization. Therefore, we add the following useful optimizations which are standard in the partial evaluation of logic programs [9]:

Delete redundant constructors: In a generalized definitional tree, the patterns of subtrees are instances of the patterns of ancestor nodes. Therefore, the generated clauses often contain redundant constructors, i.e., there are predicates p where all calls to p are of the form $p(\ldots, c(t), \ldots)$ and all left-hand sides have the same structure. In this case, we delete c.

Swap arguments for better indexing: Most Prolog implementations use first argument indexing [1]. In order to provide a portable *and* efficient implementation, we swap arguments so that the case distinction in left-hand sides is always made on the first argument (note that the *branch* nodes in a tree clearly indicate the indexed argument).

Unfold deterministic literals: The translation scheme for lazy narrowing with simplification often generates chains of predicate calls where at most one clause is applicable (see, for instance, predicates `hnf`, `simp`, `nstep`). To improve the execution time of the generated code, we unfold such deterministic predicate calls.

The optimized clauses corresponding to Example 1 can be found in Appendix C.

6 Implementation of Sharing

It is well-known that lazy evaluation strategies require the sharing of terms in order to avoid potential reevaluations of identical expressions. For instance, consider the rule

$$double(x) \rightarrow x + x$$

and the term $double(t)$ which is immediately rewritten to $t + t$. Thus, without sharing, t is evaluated twice. To avoid this problem, we have to share the result of evaluating t among the different occurrences of t. This can be implemented in Prolog by representing each function call $f(t_1, \ldots, t_n)$ by the term $f(\mathtt{S}, t_1, \ldots, t_n, \mathtt{H})$ where `S` is an unbound variable until the call $f(t_1, \ldots, t_n)$ will be evaluated (to the head normal form `H`).[7] Therefore, we only have to change the definition of the predicates which triggers the computation of a head normal form (e.g., `hnf` in Section 3) so that a term `f(S,...,H)` will be evaluated to the head normal form `H` only if `S` is an unbound variable, otherwise `H` already contains the result. Thus, the new definition of `hnf` to implement sharing is

```
hnf(T,T) :- var(T), !.
hnf(f(S,X1,...,Xn,H),H) :- !, (var(S) -> S=eval, f∅(X1,...,Xn,H)
                                ; true ).      ∀f/n ∈ F
hnf(T,T).
```

7 Experimental Results

We have implemented the translation scheme as a compiler from lazy functional logic programs into Prolog. If all functions are inductively sequential, the scheme of Section 3 is used, otherwise the scheme presented in Section 4.

First we consider inductively sequential programs. The following table contains a comparison of our translation method w.r.t. the methods proposed in [2, 6, 17, 19]. Remember that natural numbers are implemented by $0/s$-terms. The translated programs are executed with Sicstus-Prolog 2.1 on a Sparc-10. The run times are in seconds for computing the first solution (an entry "?" denotes a run time of more than 1000 seconds).

Goal:	[2]	[6]	[17]	[19]	*trans*	*sharing*	Babel	*direct*
$10000 \leq 10000 + 10000 \approx true$	0.39	6.1	0.7	0.32	0.25	0.39	0.16	0.10
$1000 \leq x + x \approx true$	3.2	86.6	?	2.7	1.9	1.8	4.3	1.2
$400 + x \leq (x + 200) + x \approx true$	4.8	?	?	2.2	1.7	2.3	4.1	0.6
$2000 \leq 1000 + (x + x) \approx true$	3.3	83.1	?	2.7	1.9	1.8	4.2	5.3
$double(double(one(100000))) \approx x$	2.8	36.1	2.9	3.5	2.8	0.9	0.35	0.17

The column *trans* contains the execution times of our translation scheme (with the optimizations of Section 5) and column *sharing* the timings of our scheme

[7] This is nearly identical to the technique proposed in [6]. Jiménez-Martin et al. [17] proposed a similar technique, but it does not really implement sharing since they omitted the evaluation flag `S`.

with sharing (Section 6). In many cases sharing has no advantage but causes an overhead (note that [2, 19] do not implement sharing). Since [6, 17] are based on narrowing strategies different from needed narrowing, the results clearly show the superiority of the needed narrowing strategy. [2] uses only one predicate to implement all rewrite rules, and Loogen et al. [19] do not perform any optimizations on the generated clauses. This explains the worse execution times in comparison to our approach.

The column "Babel" shows the execution time of needed narrowing implemented in the functional logic language Babel based on the compilation into a low-level abstract machine [11]. It is interesting to note that our high-level implementation is faster for typical search problems. The column *direct* shows the run times of a direct definition of the predicates in Prolog which is often more efficient since term structures with nested functions calls are not generated (note that *direct* corresponds to a call-by-value strategy which can be implemented more efficiently). However, there is also an example where needed narrowing is much faster since it avoids the superfluous computation of some subterms. Moreover, needed narrowing allows the computation with infinite data structures and may terminate where logic programs have an infinite search space (see, for instance, Example 1). In order to make a fair comparison between our implementation of needed narrowing and Prolog, we have omitted such examples.

The direct implementation has a good behavior on this example since current Prolog implementations are tailored towards the efficient implementation of "functional-like" programs. However, there is an interesting class of programs, namely "generate-and-test" programs, where it has been shown that narrowing with simplification can dramatically reduce the search space [8, 13]. A typical example for such programs is the "permutation sort" program, where a list is sorted by enumerating all permutations and checking whether they are sorted. In the Prolog version of this program [27, p. 55], *all* permutations are enumerated and checked. However, if we execute the same program by lazy narrowing with simplification (in this case predicates are considered as Boolean functions, see [8, p. 182]), then the simplification process cuts some parts of the search space so that not all permutations are completely enumerated. Therefore, we obtain the following execution times in seconds to sort the list `[`n`,...,2,1]` for different values of n:

Length n	Prolog	Lazy	Lazy+Simp
5	0.01	0.06	0.06
6	0.05	0.4	0.2
7	0.4	2.8	0.4
8	3.0	22.9	1.0
9	27.3	212.2	2.1
10	281.3	2188.2	4.7

The column "Lazy+Simp" contains the execution times for lazy narrowing with simplification implemented as shown in this paper, the column "Lazy" the times for pure lazy narrowing without simplification (implemented as proposed in the beginning of Section 4), and the column "Prolog" the times for the direct implementation of permutation sort in Prolog. The search spaces of "Prolog" and "Lazy" are essentially the same. However, the last column shows that the overhead of the lazy narrowing implementation can be compensated by the search space reduction due to the simplification process.

8 Conclusions

We have presented a high-level implementation of lazy functional logic languages by a transformation into Prolog. For the operational semantics, we have considered needed narrowing for inductively sequential programs and lazy narrowing with simplification for programs with overlapping left-hand sides. We have introduced generalized definitional trees in order to specify the concrete narrowing strategy. We have shown that generalized definitional trees are also useful to specify and implement the transformation of functional logic programs into Prolog. Our implementation of needed narrowing is faster compared to previous approaches, whereas the implementation of lazy narrowing with simplification is a completely new approach. We have demonstrated the advanced operational behavior of the latter strategy in comparison to Prolog for a typical class of logic programs.

Our transformation yields a portable and efficient implementation of lazy functional logic programs. Since the transformation is strongly based on the formal definition of a narrowing strategy for which soundness and completeness results are known [4, 14], the implementation is also sound and complete (modulo incompleteness problems of Prolog implementations due to the backtracking strategy). This is in contrast to other, possibly more efficient implementations of functional logic programs in Prolog with coroutining [20, 24] that do not enjoy completeness due to floundering (i.e., unevaluable delayed literals).

References

1. H. Aït-Kaci. *Warren's Abstract Machine.* MIT Press, 1991.
2. S. Antoy. Non-Determinism and Lazy Evaluation in Logic Programming. In *Proc. Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'91)*, pp. 318–331. Springer Workshops in Computing, 1991.
3. S. Antoy. Definitional Trees. In *Proc. of the 3rd Int. Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.
4. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, pp. 268–279, Portland, 1994.
5. P.G. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-Resolution. *Theoretical Computer Science 59*, pp. 3–23, 1988.
6. P.H. Cheong and L. Fribourg. Implementation of Narrowing: The Prolog-Based Approach. In *Logic programming languages: constraints, functions, and objects*, pp. 1–20. MIT Press, 1993.
7. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pp. 243–320. Elsevier, 1990.
8. L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Int. Symposium on Logic Programming*, pp. 172–184, Boston, 1985.

9. J.P. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation (PEPM'93)*, pp. 88–98. ACM Press, 1993.
10. 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.
11. W. Hans, R. Loogen, and S. Winkler. On the Interaction of Lazy Evaluation and Backtracking. In *Proc. of the 4th Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 355–369. Springer LNCS 631, 1992.
12. 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.
13. 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*, pp. 1–23. Springer LNCS 631, 1992.
14. M. Hanus. Combining Lazy Narrowing and Simplification. In *Proc. of the 6th International Symposium on Programming Language Implementation and Logic Programming*, pp. 370–384. Springer LNCS 844, 1994.
15. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
16. M. Hanus. On Extra Variables in (Equational) Logic Programming. In *Proc. International Conference on Logic Programming*, pp. 665–679. MIT Press, 1995.
17. J.A. Jiménez-Martin, J. Marino-Carballo, and J.J. Moreno-Navarro. Efficient Compilation of Lazy Narrowing into Prolog. In *Proc. Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'92)*, pp. 253–270. Springer, 1992
18. 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.
19. R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of the 5th Int. Symp. on Programming Language Implementation and Logic Programming*, pp. 184–200. Springer LNCS 714, 1993.
20. T. Mogensen. Personal Communication. 1995
21. 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.
22. 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.
23. L. Naish. *Negation and Control in Prolog*. Springer LNCS 238, 1987.
24. 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.
25. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Int. Symposium on Logic Programming*, pp. 138–151, Boston, 1985.
26. 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.
27. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

A Generated Prolog Clauses for Example 1

The program of Example 1 is inductively sequential where both functions have a unique definitional tree. Therefore, our transformation scheme of Section 3 generates the following Prolog program.

```
A===B :- hnf(A,HA), hnf(B,HB), seq(HA,HB).
seq(0,0).
seq(s(A),s(B)) :- A===B.
seq(false,false).
seq(true,true).

hnf(T,T) :- var(T), !.
hnf(A+B,H) :- !, +(A,B,H).
hnf(leq(A,B),H) :- !, leq(A,B,H).
hnf(T,T).

+(A,B,R) :- hnf(A,HA), '+_1'(HA,B,R).
'+_1'(0,B,R) :- hnf(B,R).
'+_1'(s(A),B,R) :- hnf(s(A+B),R).

leq(A,B,R) :- hnf(A,HA), leq_1(HA,B,R).
leq_1(0,B,R) :- hnf(true,R).
leq_1(s(A),B,R) :- hnf(B,HB), leq_1_2(s(A),HB,R).
leq_1_2(s(A),0,R) :- hnf(false,R).
leq_1_2(s(A),s(B),R) :- hnf(leq(A,B),R).
```

B Generated Prolog Clauses for Example 2

Since the program of Example 2 is not inductively sequential, we have to translate it by the transformation scheme of Section 4 which yields the following Prolog program.

```
A===B :- hnf(A,HA), hnf(B,HB), seq(HA,HB).
seq(0,0).
seq(s(A),s(B)) :- A===B.

hnf(T,T) :- var(T), !.
hnf(A*B,H) :- !, simp(A*B,T), nstep(T,R,_), hnf(R,H).
hnf(one(A),H) :- !, one(A,H).
hnf(T,T).

one(A,R) :- hnf(A,HA), one_1(HA,R).
one_1(0,R) :- hnf(s(0),R).
one_1(s(A),R) :- hnf(one(A),R).

simp(T,T) :- var(T), !.
simp(A*B,T) :- !, 'simp_*'(A,B,T).
simp(one(A),T) :- !, simp_one(A,T).
simp(T,T).

'simp_*'(A,B,R) :- !, simp(A,SA),
    (nonvar(SA) -> 'simp_*_1'(SA,B,R) ; 'simp_*_or'(SA,B,R)).
'simp_*_1'(0,A,R) :- !, simp(0,R). % first alternative of *
'simp_*_1'(A,B,R) :- 'simp_*_or'(A,B,R).
```

```
'simp_*_or'(A,B,R) :- !, simp(B,SB),
    (nonvar(SB) -> 'simp_*_or_2'(A,SB,R) ; R=A*SB).
'simp_*_or_2'(A,0,R) :- !, simp(0,R). % second alternative of *
'simp_*_or_2'(A,B,A*B).

simp_one(A,R) :- !, simp(A,SA),
    (nonvar(SA) -> simp_one_1(SA,R) ; R=one(SA)).
simp_one_1(0,R) :- !, simp(s(0),R).
simp_one_1(s(A),R) :- !, simp(one(A),R).
simp_one_1(A,one(A)).

nstep(T,T,C) :- var(T), !.
nstep(A*B,T,C) :- !, '*_step'(A,B,T,C).
nstep(one(A),T,C) :- !, one_step(A,T,C).
nstep(T,T,C).

'*_step'(A,B,R,C) :- nstep(A,NA,C),
    (var(C) -> '*_step_1'(NA,B,R,C) ; R=NA*B).
'*_step'(A,B,R,C) :- nstep(B,NB,C),
    (var(C) -> '*_step_2'(A,NB,R,C) ; R=A*NB).
'*_step_1'(0,A,0,step).
'*_step_2'(A,0,0,step).

one_step(A,R,C) :- nstep(A,NA,C),
    (var(C) -> one_step_1(NA,R,C) ; R=one(NA)).
one_step_1(0,s(0),step).
one_step_1(s(A),one(A),step).
```

C Optimized Prolog Program for Example 1

If we apply the optimization techniques discussed in Section 5 to the program of Appendix A, we obtain the following optimized Prolog program (where superfluous clauses are deleted).

```
A===B :- hnf(A,HA), hnf(B,HB), seq(HA,HB).
seq(0,0).
seq(s(A),s(B)) :- hnf(A,HA), hnf(B,HB), seq(HA,HB).
seq(false,false).
seq(true,true).

hnf(T,T) :- var(T), !.
hnf(A+B,H) :- !, hnf(A,HA), '+_1'(HA,B,H).
hnf(leq(A,B),H) :- !, hnf(A,HA), leq_1(HA,B,H).
hnf(T,T).

'+_1'(0,B,R) :- hnf(B,R).
'+_1'(s(A),B,s(A+B)).

leq_1(0,B,true).
leq_1(s(A),B,R) :- hnf(B,HB), leq_1s_2(HB,A,R).
leq_1s_2(0,A,false).
leq_1s_2(s(B),A,R) :- hnf(A,HA), leq_1(HA,B,R).
```