A Call-by-Need Strategy for Higher-Order Functional-Logic Programming

Christian Prehofer
Technische Universität München
prehofer@informatik.tu-muenchen.de

Abstract

We present an approach to truely higher-order functional-logic programming based on higher-order narrowing. Roughly speaking, we model a higher-order functional core language by higher-order rewriting and extend it by logic variables. For the integration of logic programs, conditional rules are supported. For solving goals in this framework, we present a complete calculus for higher-order conditional narrowing. We develop several refinements that utilize the determinism of functional programs. These refinements can be combined to a narrowing strategy which generalizes call-by-need as in functional programming, where the dedicated higher-order methods are only used for full higher-order goals. Furthermore, we propose an implementational model for this narrowing strategy which delays computations until needed.

1 Introduction

We present a novel approach towards the integration of higher-order functional and logic programming (for a survey see [6]). The goal was to design a simple language, in contrast to a language that subsumes both. The primary feature we support is higher-order programming, which is common in functional languages, but not in (functional-)logic programming [6].

Roughly speaking, we extend a higher-order functional core language by logic variables as in Prolog. These logic variables may be higher-order, which is hard to avoid in this context. Thus we need higher-order unification, as e.g. in λProlog. The language is based on higher-order rewrite rules, which model functional programming (and actually more). To support logic programming, we allow conditions with extra variables.

Our higher-order setting allows for highly expressive constructs, e.g. symbolic differentiation. The function $diff(F, X)$, defined by

$$\begin{aligned} diff(\lambda y.F, X) &\rightarrow 0 \\ diff(\lambda y.y, X) &\rightarrow 1 \\ diff(\lambda y.sin(F(y)), X) &\rightarrow cos(F(X)) * diff(\lambda y.F(y), X) \\ diff(\lambda y.ln(F(y)), X) &\rightarrow diff(\lambda y.F(y), X)/F(X), \end{aligned}$$

computes the differential of a function F at a point X. With these rules, we

can not only evaluate, as e.g.,

$$diff(\lambda y.sin(sin(y)), X) \xrightarrow{*} cos(sin(X)) * cos(X)$$

but also solve goals modulo these rules, as shown later.

On the technical side, we contribute the following:

- Completeness results for conditional higher-order narrowing.
- A call-by-need narrowing strategy, motivated by call-by-need in functional programming, which utilizes properties of functional programs.
- A simpler operational model with leads to decidable higher-order unification in the second-order case, i.e. a program may not diverge only due to unification.

We use directed equational goals of the form the form $s \rightarrow^{?} t$, where θ is a solution if $\theta s \xrightarrow{*} \theta t$. Intuitively, the computation in such goals proceeds from left to right. Our approach admits higher-order rules $l \rightarrow r \Leftarrow c \rightarrow c_{gr}$, where c_{gr} is a ground (or closed) term in normal form and c may have extra variables not occurring in l. Then for solving conditions of rules, as well as for queries, oriented goals suffice. The restriction to ground right-hand sides in the conditions also simplifies the technical treatment and helps to establish confluence/termination (see e.g. [7]). We argue that these restrictions do not impede programming applications. In our higher-order functional setting, extra variables on the right are not needed, since we may use functional *let* or *where* constructs, as shown later.

2 Preliminaries

We briefly introduce simply typed λ-calculus (see e.g. [8]). We assume the following **variable conventions**:

- F, G, H, X, Y denote free variables,
- a, b, c, f, g (function) constants, and
- x, y, z bound variables.

Type judgments are written as $t : \tau$. Further, we often use s and t for terms and u, v, w for constants or bound variables. The set of types for the simply typed λ-terms is generated by a set of base types (e.g. int, bool) and the function type constructor $\rightarrow$. The syntax for **λ-terms** is given by

$$t \quad = \quad F \mid x \mid c \mid \lambda x.t \mid (t_1\ t_2)$$

A list of syntactic objects $s_1, \ldots, s_n$ where $n \geq 0$ is abbreviated by $\overline{s_n}$. For instance, n-fold abstraction and application are written as $\lambda\overline{x_n}.s = \lambda x_1 \ldots \lambda x_n.s$ and $a(\overline{s_n}) = ((\cdots(a\ s_1)\cdots)\ s_n)$, respectively. Free and bound variables of a term t will be denoted as $\mathcal{FV}(t)$ and $\mathcal{BV}(t)$, respectively. Let $\{x \mapsto s\}t$ denote the result of replacing every free occurrence of x in t by s. Besides α-conversion, i.e. the consistent renaming of bound variables, the **conversions in λ-calculus** are defined as:

- β**-conversion:** $(\lambda x.s)t =_\beta \{x \mapsto t\}s$, and
- η**-conversion:** if $x \notin \mathcal{FV}(t)$, then $\lambda x.(tx) =_\eta t$.

The long $\beta\eta$-normal form of a term t, denoted by $t\downarrow^\eta_\beta$, is the η-expanded form of the $\beta\eta$-normal form of t. It is well known [8] that $s =_{\alpha\beta\eta} t$ iff $s\downarrow^\eta_\beta =_\alpha t\downarrow^\eta_\beta$. As long $\beta\eta$-normal forms exist for typed λ-terms, we will in general assume that terms are in long $\beta\eta$-normal form. For brevity, we may write variables in η-normal form, e.g. X instead of $\lambda\overline{x_n}.X(\overline{x_n})$. We assume that the transformation into long $\beta\eta$-normal form is an implicit operation, e.g. when applying a substitution to a term.

A substitution θ is in long $\beta\eta$-normal form if all terms in the image of θ are in long $\beta\eta$-normal form. The convention that α-equivalent terms are identified and that free and bound variables are kept disjoint (see also [4]) is used in the following. Furthermore, we assume that bound variables with different binders have different names. Define $\mathcal{D}om(\theta) = \{X \mid \theta X \neq X\}$ and $\mathcal{R}ng(\theta) = \bigcup_{X \in \mathcal{D}om(\theta)} \mathcal{FV}(\theta X)$. Two **substitutions are equal on a set of variables** W**,** written as $\theta =_W \theta'$, if $\theta\alpha = \theta'\alpha$ for all $\alpha \in W$. A substitution θ is **idempotent** iff $\theta = \theta\theta$. We will in general assume that substitutions are idempotent. A substitution θ' is more general than θ, written as $\theta' \leq \theta$, if $\theta = \sigma\theta'$ for some substitution σ.

We describe positions in λ-terms by sequences over natural numbers. The subterm at a **position** p in a λ-term t is denoted by $t|_p$. A term t with the subterm at position p replaced by s is written as $t[s]_p$.

A term t in β-normal form is called a (**higher-order**) **pattern** if every free occurrence of a variable F is in a subterm $F(\overline{u_n})$ of t such that the $\overline{u_n}$ are η-equivalent to a list of distinct bound variables. Unification of patterns is decidable and a most general unifier exists if they are unifiable [13]. Also, the unification of a linear pattern with a second-order term is decidable and finitary, if they are variable-disjoint [16]. Examples of higher-order patterns are $\lambda x, y.F(x, y)$ and $\lambda x.f(G(\lambda z.x(z)))$, where the latter is at least third-order. Non-patterns are for instance $\lambda x, y.F(a, y)$ and $\lambda x.G(H(x))$.

A **rewrite rule** [15, 12] is a pair $l \to r$ such that l is a pattern but not a free variable, l and r are long $\beta\eta$-normal forms of the same base type, and $\mathcal{FV}(l) \supseteq \mathcal{FV}(r)$. Assuming a rule $l \to r$ and a position p in a term s in long $\beta\eta$-normal form, a **rewrite step** from s to t is defined as

$$s \longrightarrow^{l \to r}_{p,\theta} t \quad \Leftrightarrow \quad s|_p = \theta l \ \wedge \ t = s[\theta r]_p.$$

For a rewrite step we often omit some of the parameters $l \to r, p$ and θ. We assume that constant symbols are divided into free **constructor symbols** and defined symbols. A symbol f is called a **defined symbol**, if a rule $f(\ldots) \longrightarrow t$ exists. Constructor symbols are denoted by c and d. A term is in R**-normal form** for a set or rewrite rules R if no rule from R applies and a substitution θ is R**-normalized** if if all terms in the image of θ are in R-normal form.

Notice that a subterm $s|_p$ may contain free variables which used to be bound in s. For rewriting it is possible to ignore this, as only matching of a left-hand side of a rewrite rule is needed. For narrowing, we need unification and hence we use the following construction to lift a rule into a binding context to facilitate the technical treatment.

An $\overline{x_k}$**-lifter** of a term t **away from** W is a substitution $\sigma = \{F \mapsto (\rho F)(\overline{x_k}) \mid F \in \mathcal{FV}(t)\}$ where ρ is a renaming such that $\mathcal{D}om(\rho) = \mathcal{FV}(t)$, $\mathcal{R}ng(\rho) \cap W = \{\}$ and $\rho F : \tau_1 \to \cdots \to \tau_k \to \tau$ if $x_1 : \tau_1, \ldots, x_k : \tau_k$ and $F : \tau$. A term t (rewrite rule $l \to r$) is $\overline{x_k}$-lifted if an $\overline{x_k}$-lifter has been applied to t (l and r). For example, $\{X \mapsto X'(x)\}$ is an x-lifter of $g(X)$ away from any W not containing X'

3 Conditional Lazy Narrowing

In this section, we propose a class of conditional higher-order rewrite rules which are tailored for functional programming languages. Then we introduce a system of transformations for this class of rules. Further optimizations are developed in later sections.

Definition 3.1 A **normal conditional higher-order rewrite system (NCHRS)** R is a set of conditional rewrite rules of the form $l \to r \Leftarrow \overline{l_n \to r_n}$, where $l \to r$ is a rewrite rule and $\overline{r_n}$ are ground R-normal forms. A **conditional rewrite step** is defined as $s \longrightarrow_{p,\theta}^{l \to r \Leftarrow \overline{l_n \to r_n}} t$ iff $s \longrightarrow_{p,\theta}^{l \to r} t$ and $\overline{\theta l_n \xrightarrow{*}{}^{R} r_n}$.

Notice that rewrite rules are restricted to base type, but the conditions may be higher-order. Also, oriented goals suffice for proving the conditions as $\overline{\theta l_n \overset{*}{\longleftrightarrow} \theta r_n}$ is equivalent with $\overline{\theta l_n \xrightarrow{*} r_n}$.

The rules of System CLN for lazy higher-order narrowing are shown in Figure 1. The rules are split into standard (first-order) rules, plus higher-order rules. These consist of a rule for narrowing at variables, needed to compute functional objects, and rules for higher-order unification. The higher-order rules will only be needed if truly higher-order free variables occur (in non pattern terms, to be precise). For brevity, some type constraints of the rules, which particularly restrict the higher-order rules, are left implicit.

Let $s \overset{?}{\leftrightarrow} t$ stand for one of $s \to^? t$ and $t \to^? s$. For goals of the form $s \overset{?}{\leftrightarrow} t$, the rules are intended to preserve the orientation of $\overset{?}{\leftrightarrow}$. We extend the transformation rules on goals to sets of goals in the canonical way: $\{\overline{s \to^? t}\} \cup S \Rightarrow^\theta \{\overline{s_n \to^? t_n}\} \cup \theta S$ if $s \to^? t \Rightarrow^\theta \{\overline{s_n \to^? t_n}\}$. For a sequence $\Rightarrow^{\theta_1} \ldots \Rightarrow^{\theta_n}$ of CLN steps, we write $\overset{*}{\Rightarrow}{}^\theta$, where $\theta = \theta_n \ldots \theta_1$. Goals of the form $\lambda\overline{x_k}.F(\ldots) \overset{?}{\leftrightarrow} \lambda\overline{x_k}.G(\ldots)$, called **flex-flex**, are guaranteed to have some solution and are usually delayed in higher-order unification.

The main ingredient for completeness of conditional narrowing is to assure that solutions for fresh variables in the conditions are normalized. This

Decomposition

$$\lambda\overline{x_k}.v(\overline{t_n}) \to^? \lambda\overline{x_k}.v(\overline{t'_n}) \quad \Rightarrow \quad \{\overline{\lambda\overline{x_k}.t_n \to^? \lambda\overline{x_k}.t'_n}\}$$

Elimination

$$F \stackrel{?}{\leftrightarrow} t \quad \Rightarrow^\theta \quad \{\} \text{ if } F \notin \mathcal{FV}(t) \text{ and where } \theta = \{F \mapsto t\}$$

Conditional Narrowing with Decomposition

$$\lambda\overline{x_k}.f(\overline{t_n}) \to^? \lambda\overline{x_k}.t \quad \Rightarrow \quad \{\overline{\lambda\overline{x_k}.t_n \to^? \lambda\overline{x_k}.l_n},\ \overline{\lambda\overline{x_k}.l'_p \to^? \lambda\overline{x_k}.r'_p},\ \lambda\overline{x_k}.r \to^? \lambda\overline{x_k}.t\}$$
where $f(\overline{l_n}) \to r \Leftarrow \overline{l'_p \to r'_p}$ is an $\overline{x_k}$-lifted rule

---

Truly Higher-Order Rules

Conditional Narrowing at Variable

$$\lambda\overline{x_k}.H(\overline{t_n}) \to^? \lambda\overline{x_k}.t \quad \Rightarrow^\theta \quad \{\overline{\lambda\overline{x_k}.H_m(\overline{\theta t_n}) \to^? \lambda\overline{x_k}.l_m},\ \overline{\lambda\overline{x_k}.l'_p \to^? \lambda\overline{x_k}.r'_p},\ \lambda\overline{x_k}.r \to^? \lambda\overline{x_k}.t\}$$
if $\lambda\overline{x_k}.H(\overline{t_n})$ is not a pattern,
$f(\overline{l_m}) \to r \Leftarrow \overline{l'_p \to r'_p}$ is an $\overline{x_k}$-lifted rule,
and $\theta = \{H \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\}$
with fresh variables $\overline{H_m}$

Imitation

$$\lambda\overline{x_k}.F(\overline{t_n}) \stackrel{?}{\leftrightarrow} \lambda\overline{x_k}.f(\overline{t'_m}) \quad \Rightarrow^\theta \quad \{\overline{\lambda\overline{x_k}.H_m(\overline{\theta t_n}) \stackrel{?}{\leftrightarrow} \lambda\overline{x_k}.t'_m}\}$$
where $\theta = \{F \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\}$
with fresh variables $\overline{H_m}$

Projection

$$\lambda\overline{x_k}.F(\overline{t_n}) \stackrel{?}{\leftrightarrow} \lambda\overline{x_k}.v(\overline{t'_m}) \quad \Rightarrow^\theta \quad \{\lambda\overline{x_k}.\theta t_i(\overline{H_p(\overline{t_n})}) \stackrel{?}{\leftrightarrow} \lambda\overline{x_k}.v(\overline{t'_m})\}$$
where $\theta = \{F \mapsto \lambda\overline{x_n}.x_i(\overline{H_p(\overline{x_n})})\}$,
$\overline{H_p : \tau_p}$, and $x_i : \overline{\tau_p} \to \tau$
with fresh variables $\overline{H_p}$

Figure 1: System CLN for Conditional Lazy Narrowing

is possible for extra variables on the left if the system is convergent or at least confluent and weakly normalizing (which means that normal forms exist) as above. This is the reason for disallowing extra variables on the right.

Theorem 3.2 *[Completeness of CLN] Assume a confluent and weakly normalizing NCHRS R. If $s \to^? t$ has solution θ, i.e. $\theta s \xrightarrow{*}{}^R \theta t$, θt and θ are R-normalized, then $\{s \to^? t\} \Rightarrow^{\delta}_{CLN} F$ such that δ is more general, modulo the newly added variables, than θ and F is a set of flex-flex goals.*[1]

3.1 Refinements Using the Determinism of Functional Languages

We mention briefly some important refinements that have been established in the higher-order setting [18, 19].

- Simplification, i.e. (partial) functional evaluation, has shown to be complete for convergent systems.
- Binding variables via variable elimination is a very natural rule, but in general its completeness is an open problem. In our context, elimination on goals $X \to^? t$ is complete.
- It is useful to add some refinements for constructors. First, decomposition on constructors, e.g. on $c(\ldots) \to^? c(\ldots)$ is deterministic. Correspondingly, goals of the form $c(\ldots) \to^? v(\ldots)$, where v is not a variable and $v \neq c$ are unsolvable, since evaluation proceeds from left to right.

4 Left-Linear Programs and Simple Systems

In this section we examine a particular class of goal systems, Simple Systems [17], which suffice for programming and have several interesting properties. We assume in the following NCHRS with left-linear rules. A rule $l \to r \Leftarrow \overline{l_n \to r_n}$ is **left-linear**, if no free variable occurs repeatedly in l.

Definition 4.1 We write $s \to^? s' \ll t \to^? t'$, if $\mathcal{FV}(s') \cap \mathcal{FV}(t) \neq \{\}$.

This order links goals by the variables occurring, e.g. $t \to^? f(X) \ll X \to^? s$. Now we are ready to define Simple Systems:

Definition 4.2 A system of goals $\overline{G_n} = \overline{s_n \to^? t_n}$ is a **Simple System**, if

- all right-hand sides $\overline{t_n}$ are patterns,
- $\overline{G_n}$ is cycle free, i.e. the transitive closure of $\ll$ is a strict partial ordering on $\overline{G_n}$ and
- every variable occurs at most once in the right-hand sides $\overline{t_n}$.

We show next that this class is closed under the rules of CLN.

Theorem 4.3 *Assume a left-linear NCHRS R. If $\overline{G_n}$ is a Simple System, then applying CLN with R preserves this property.*

The following results on solved forms from [17] will be crucial later.

[1] Detailed proofs can be found in [19].

Theorem 4.4 *A Simple System* $S = \{X_1 \stackrel{?}{\leftrightarrow} t_1, \ldots, X_n \stackrel{?}{\leftrightarrow} t_n\}$ *has a solution if all* $\overline{X_n}$ *are distinct.*

The following corollary is needed for the new narrowing strategy developed later.

Corollary 4.5 *A Simple System of the form* $\{\overline{t_n \rightarrow^? X_n}\}$ *is solvable.*

In the second-order case unification never leads to divergence, as shown in [17], extending results in [16]:

Theorem 4.6 *Solving a second-order Simple System by the unification rules of CLN, i.e. without the narrowing rules, terminates.*

5 Variables of Interest

In the following, we classify variables in Simple Systems into variables of interest and intermediate variables. This prepares the narrowing strategy presented in the next section.

We consider initial goals of the form $s \rightarrow t$, and assume that only the values for the free variables in s are of interest, neither the variables in t nor intermediate variables computed by CLN. For instance, assume the rule $f(a, X) \rightarrow g(b)$ and the goal $f(Y, t) \rightarrow^? g(b)$, which is transformed to

$$Y \rightarrow^? a, t \rightarrow^? X, g(b) \rightarrow^? g(b)$$

by Lazy Narrowing. Clearly, only the value of Y is of interest for solving the initial goal, but not the value of X.

The invariant we will show is that variables of interest only occur on the left, but never on the right-hand side of a goal. We first need to define the notion of variables of interest. Consider an execution of CLN. We start with a goal $s \rightarrow^? t$ where initially the variables of interest are in s. This has to be updated for each CLN step. If X is a variable of interest, and an CLN step computes δ, then the free variables in δX are the new variables of interest. With this idea in mind we define the following:

Definition 5.1 Assume a sequence of transformations $\{s \rightarrow^? t\} \stackrel{*}{\Rightarrow}^{\delta}_{CLN} \{\overline{s_n \rightarrow^? t_n}\}$. A variable X is called a **variable of interest** if $X \in \mathcal{FV}(\delta s)$ and intermediate otherwise.

Now we can show the following result:

Theorem 5.2 *Assume a left-linear NCHRS* R*, a Simple System* $\overline{G_n} = \{\overline{s_n \rightarrow^? t_n}\}$ *and a set of variables* V *with* $V \cap \mathcal{FV}(\overline{t_n}) = \{\}$*. If* $\overline{G_n} \Rightarrow^{\delta}_{CLN} \{\overline{s'_m \rightarrow^? t'_m}\}$*, then* $((V - \mathcal{D}om(\delta)) \cup \mathcal{R}ng(\delta)) \cap \mathcal{FV}(\overline{t'_m}) = \{\}$*.*

Then the desired result follows easily:

Corollary 5.3 (Variables of Interest) *Assume a left-linear NCHRS* R *and assume solving a Simple System* $s \rightarrow^? t$ *with system CLN. Then variables of interest only occur on the left, but never on the right-hand side of a goal.*

6 Call-By-Need Narrowing

We show that for Simple Systems a strategy for variable elimination leads to a new narrowing strategy, coined call-by-need narrowing. In essence, we show that certain goals can safely be delayed, which means that computations are only performed when needed.

As we consider oriented equations, we can distinguish two cases of variable elimination and we will handle variable elimination appropriately in each case. In the first case, $X \to^? t$, the variable X can be a variable of interest. Thus the elimination of X is desirable for computational reasons and is deterministic (Sec. 3.1). Notice that elimination is always possible on such goals, as $X \notin \mathcal{FV}(t)$ in Simple Systems.

In the other case of variable elimination, i.e.

$$t \to^? X,$$

elimination may not be deterministic. Thus such goals will be delayed. This simple strategy has some interesting properties, which we will examine in the following.

First view this idea in the context of a programming language. Let us for instance model the evaluation (or normalization) $f(t_1, t_2)\downarrow_R = t$ by narrowing, assuming the rule $f(X, Y) \to g(X, X)$:

$$\{f(t_1, t_2) \to^? t\} \Rightarrow_{CLN} \{t_1 \to^? X, t_2 \to^? Y, g(X, X) \to^? t\}$$

Now we can model the following evaluation strategies:

Eager evaluation (or call-by-value) is obtained by performing normalization on the goals t_1 and t_2, followed by eager variable elimination on $t_1\downarrow_R \to^? X$ and $t_2\downarrow_R \to^? Y$. The disadvantage is that eager evaluation may perform unnecessary evaluation steps.

Call-by-name is obtained by immediate eager variable elimination on $t_1 \to^? X$ and on $t_2 \to^? Y$. It has the disadvantage that terms are copied, e.g. t_1 here as X occurs twice in $g(X, X)$. Thus expensive evaluation may have to be done repeatedly.

Needed evaluation (or call-by-need) is an evaluation strategy that can be obtained by delaying the goals $t_1 \to^? X$ and $t_2 \to^? Y$, thus avoiding copying. Then t_1 and t_2 are only evaluated when X or Y are needed for further computation.

In the latter, we model equationally lazy evaluation with sharing copies of identical subterms, i.e. the delayed equations may be viewed as shared subterms. The notion of need considered here is similar to the notion of call-by-need in [2], but not to optimal or needed reduction [9].

Let us now come back from evaluation to the context of narrowing. Consider for instance the narrowing step with the above rule

$$\{f(t_1, t_2) \to^? g(a, Z)\} \Rightarrow_{CLN} \{t_1 \to^? X, t_2 \to^? Y, g(X, X) \to^? g(a, Z)\}$$

In contrast to evaluation as in functional languages, solving the goals $t_1 \to^? X, t_2 \to^? Y$ may have many solutions. Whereas in functional languages, eager evaluation can be faster, this is unclear for functional-logic programming. Thus we suggest to adopt the following "call-by-need" approach:

Definition 6.1 Call-By-Need Narrowing is defined as Lazy Narrowing with System CLN where goals of the form $t \to^? X$ are delayed.

For instance, in the above example, decomposition on $g(X, X) \to^? g(a, Z)$ yields the goals $X \to^? a, X \to^? Z$. Deterministic elimination on $X \to^? a$ instantiates X, thus the goal $t_1 \to^? a$ has to be solved, i.e. a valued for t_1 is needed. In contrast, $t_2 \to^? Y$ is delayed.

This new notion of narrowing for Simple Systems and left-linear NCHRS is supported by the following arguments: **Call-By-Need Narrowing**

is complete, or safe, in the sense that when only goals of the form $\overline{t_n \to^? X_n}$ remain, they are solvable by Corollary 4.5. Since the strategy is to delay such goals, this result is essential.[2]

delays intermediate variables only. As shown in the last section, we can identify the variables to be delayed: a variable X in a goal $t \to^? X$ cannot be a variable of interest.

avoids copying, as shown above, variable elimination on intermediate variables possibly copies unevaluated terms and duplicates work. Thus intermediate goals of the form $t \to^? X$ are only considered if X is instantiated, i.e. if a value is needed.

The important aspect of this strategy is that the higher-order rules are only needed if higher-order free variables occur; goals with a first-order variable on one side are either solved by elimination, as the occurs check is immaterial, or simply delayed.

The analogy to call-by-need in programming languages leads to another simple improvement. On a goal of the form $X(t) \to^? t'$ the higher-order rules have to be applied in general. However, if a goal $s \to^? X$ exists, then it is advantageous to compute a value for X from this goal before attempting the higher-order rules. This case is particularly easy to detect if delayed goals are viewed as a context, which we show next.

[2] This may conflict with flex-flex pairs in some special cases [19].

6.1 Implementation Considerations

This section discusses more operational and implementational aspects of call-by-need narrowing. In the following abstract model for call-by-need narrowing, goals are delayed in a context after Decomposition and Lazy Narrowing and are possibly reactivated by Elimination. The idea is to handle intermediate evaluations effectively and to detect deterministic operations on-the-fly.

The important step is to view the delayed goals for call-by-need narrowing as a "context" and to consider an intermediate variable as a pointer to a delayed term. This is possible for the following two reasons: intermediate variables can be characterized and, more importantly, variables can occur only once on the right and can thus be seen as a pointer to a (single!) term. Thus we get contexts for free, i.e. we do not need any extra machinery.

Assume a set of delayed goals, or a context,

$$G_d = \overline{t_n \to^? X_n},$$

where $\overline{X_n}$ are guaranteed to be distinct, and a set of active goals

$$G_a = \overline{s_m \to^? s'_m}.$$

For an implementation, we assume that the intermediate variables $\{\overline{s_m}\} \cap \{\overline{X_n}\}$ have a "pointer" to their delayed goal in G_d. (Unfortunately the arrow in a delayed goal $t \to^? X$ gives the wrong direction for viewing this as a pointer.)

With this model in mind, we first examine the first-order rules from CLN on a goal from G_a. The idea of the following is to scan newly generated goals on-the-fly for deterministic operations.

Elimination on a goal $X \to^? t$ performs a "wake-up" on a delayed goal $t_i \to^? X$, if $X \in \{\overline{X_n}\}$.

Decomposition on a goal $v(\overline{t_p}) \to^? v(\overline{t'_p})$ creates the new goals $\overline{t_p \to^? t'_p}$.

Conditional Narrowing on a goal $f(\overline{t_p}) \to^? s$ with a rule $f(\overline{l_p}) \to r \Leftarrow \overline{l_o \to r_o}$ creates the new goals $\overline{t_p \to^? l_p}, \overline{l_o \to^? r_o}, r \to^? s$.

For a set of new goals $\overline{G_n} = \overline{t_p \to^? t'_p}$, created by the Decomposition or Narrowing rule, we examine if a deterministic operation is possible (see Sec. 3.1) and if the goal is to be delayed. A goal from $\overline{G_n}$ can be of one of the following forms:

1. $u(\overline{t_k}) \to^? v(\overline{t'_l})$
2. $X \to^? v(\overline{t'_k})$
3. $u(\overline{t_k}) \to^? X$

When creating these goals, we check for deterministic simplification as follows. For the first form, we only check if a deterministic decomposition or if a constructor clash applies. Elimination is performed on goals of the second form. This may reactivate a delayed goal which is added to the new goals $\overline{G_n}$ and is checked as well. In the remaining case, goals of the third form are delayed. This (recursive) simplification procedure must terminate, as we only perform unification rules.

For the higher-order rules, we cannot hope for much preprocessing as above. Imitation and Conditional Narrowing at Variable on a goal create new goals with variable heads where in some cases Projection is the only operation that applies.

7 Examples

This section presents examples for higher-order functional-logic programming; more example can be found in [19].

Strict equality on first-order data types is common in functional(-logic) programming languages. With strict equality $=_s$ two terms are equal, if they can be evaluated to the same (constructor) term. It is interesting to see how strict equality can be encoded in our setting. For instance, the rules

$$\begin{array}{rcl} s(X) =_s s(Y) & \to & X =_s Y \\ 0 =_s 0 & \to & true \end{array}$$

suffice for the constructors s and 0. With strict equality, we can avoid full equality on higher-order terms, similar to current functional(-logic) languages. Recall that full equality entails undecidable second-order unification.

7.1 Computing Ancestor Relations

This example computes ancestor relations from a simple database. In the first-order case, such examples are used to find persons that are related in some way, here we can also compute the relation explicitly as a λ-term.

$$\begin{array}{lcl} map(F,[X|Y]) & \to & [F(X)|map(F,Y)] \\ map(F,[]) & \to & [] \\ father(mary) & \to & john \\ father(john) & \to & art \end{array}$$

With these rules, the query

$$R(mary) \to^? art$$

has the solution $R \mapsto \lambda x.father(father(x))$ and the goal

$$map(F,[mary,john]) \to^? [john,art]$$

is solved by $F \mapsto \lambda x.father(x)$.

7.2 Symbolic Differentiation

Using the rules for differentiation of Section 1, we can solve the following goal by call-by-need narrowing. For simplicity, we also use functional evaluation [18] in this example.

$$
\begin{array}{ll}
\{\lambda x.diff(\lambda y.ln(F(y)),x) \to^? \lambda x.cos(x)/sin(x)\} & \stackrel{*}{\Rightarrow} \text{Evaluation for } diff \\
\{\lambda x.diff(\lambda y.F(y),x)/F(x) \to^? \lambda x.cos(x)/sin(x)\} & \stackrel{*}{\Rightarrow} \text{Decomposition} \\
\{\lambda x.diff(\lambda y.F(y),x) \to^? \lambda x.cos(x), & \\
\ \lambda x.F(x) \to^? \lambda x.sin(x)\} & \stackrel{*}{\Rightarrow} \text{Elimination} \\
\{\lambda x.diff(\lambda y.sin(y),x) \to^? \lambda x.cos(x)\} & \stackrel{*}{\Rightarrow} \text{Evaluation} \\
\{\lambda x.cos(x) * diff(\lambda y.y,x) \to^? \lambda x.cos(x)\} & \stackrel{*}{\Rightarrow} \text{Evaluation} \\
\{\lambda x.cos(x) \to^? \lambda x.cos(x)\} & \stackrel{*}{\Rightarrow} \text{Decomposition} \\
\{\} &
\end{array}
$$

Compared to [17], there is no search necessary to find the solution $F \mapsto \lambda x.sin(x)$ by the call-by-need strategy.

8 Discussion and Related Work

In this section, we briefly discuss some important aspects of this approach: the restrictions imposed on conditions and, secondly, a comparison to an optimal first-order strategy.

We argue that in the higher-order case extra variables in right side of the conditions are not needed for programming purposes. Whereas in (functional-)logic programming such extra variables are often used as local variables, we prefer the more suitable constructs of functional programming here. Consider for instance the function *unzip*, splitting a list of pairs into a pair of lists, which we write in a functional way:

$$unzip([(x,y)|R]) \to \mathsf{let}\ (xs,ys) = unzip(R)\ \mathsf{in}\ ([x|xs],[y|ys])$$

This is usually written as $unzip([(x,y)|R]) \to ([x|xs],[y|ys]) \Leftarrow unzip(R) \to (xs,ys)$ in first-order languages, which requires extra variables on the right. The above notation for a let-construct corresponds to

$$\mathsf{let}\ (xs,ys) = X\ \mathsf{in}\ F(xs,ys) =^{def} let\ X\ in\ \lambda xs,ys.F(xs,ys)$$

which can be defined by a higher-order rewrite rule

$$let\ (Xs,Ys)\ in\ \lambda xs,ys.F(xs,ys) \to F(Xs,Ys).$$

On the other hand, we use existential logic variables in conditions for relational programming, e.g. a grandmother predicate:

$$grand\_mother(X,Y) \Leftarrow mother(X,Z), mother(Z,Y)$$

Next we compare this approach to some advanced first-order ones. For a restricted class of rewrite rules, i.e. inductively sequential, an optimal narrowing strategy, called needed narrowing, has been presented in [1]. As in other first-order approaches to functional-logic programming, an alternative definition of narrowing is used: we write $t \leadsto t'$ for a narrowing step if some subterm $t|_p$ is unified with the left-hand side of a rule such that θt can be rewritten to t'. For this notion of narrowing many refinements have been developed, but in the general higher-order case this approach has some principal problems with bound variables [17]. It is thus not clear how to compare these two approaches. Generally, needed narrowing in [1] is optimal wrt the length of the reduction steps performed (modulo sharing; for a precise definition see [9]).

One difference is that we have a clear model to prefer deterministic operations. For instance, with the following rules

$$\begin{array}{lcl} f(0) & \to & 0 \\ ones(0) & \to & s(0) \\ ones(s(X)) & \to & ones(X) \end{array}$$

the goal $f(ones(X)) \to^? s(0)$ obviously has no solution. This is detected here by a constructor clash during simplification (see Sec. 6.1). On the other hand, a strategy driven purely by optimal reductions, such as in [1], attempts narrowing steps at the inner $ones(X)$ redex and diverges. Although it is difficult to compare these approaches for practical applications, notice that Prolog, when coding the above into predicates, performs the same simplification, i.e. unification fails.

A disadvantage of the outside-in approach of lazy narrowing is that redundant computations in different search trees are possible. For instance, consider the goal $ones(t) \to^? s(Y)$, where t is an arbitrary term. In this example, for each of the two rules, the term t has to be evaluated, i.e. in two goals $t \to^? 0$ and $t \to^? s(X)$. This can be avoided by needed narrowing [1]. Again, a naive translation into Prolog may exhibit the same inefficiency, depending on the order of the literals. Also, this may not be a problem for mostly functional programs with little branching.

9 Conclusions

We have presented an effective model for the integration of functional and logic programming. We have shown that the restrictions in our setting, motivated by functional programming, lead to operational benefits and to a call-by-need narrowing strategy. A particular feature is that truly higher-order goals with higher-order logic variables are solved by distinguished rules.

In contrast to many other works on higher-order functional-logic programming [5, 11, 21], we cover the full higher-order case. The work in [20] on higher-order narrowing considers only a restricted class of λ-terms,

higher-order patterns with first-order equations, which does not suffice for modeling higher-order functional programs. The approach to higher-order narrowing in [3] aims at narrowing with higher-order functional programs, but restricts higher-order variables in the left-hand sides of rules and only permits restricted goals.

Compared to higher-order logic programming [14], higher-order programming as in functional languages is possible directly here. Our results may also contribute to (operational) semantics of the language Escher [10], which pursues similar goals as done here.

Acknowledgements The author is grateful to the helpful comments of the anonymous referees. This research was supported by the DFG under grant Br 887, *Deduktive Programmentwicklung* and by ESPRIT WG 6028, *CCL*.

References

[1] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994.

[2] Zena Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *22'nd ACM Symposium on Principles of Programming Languages*, San Francisco, California, 1995.

[3] J. Avenhaus and C. A. Loría-Sáenz. Higher-order conditional rewriting and narrowing. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, München, Germany, September 1994. Springer LNCS 845.

[4] Hendrik Pieter Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North Holland, 2nd edition, 1984.

[5] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. On the completeness of narrowing as the operational semantics of functional logic programming. In E. Börger, G. Jäger, H. Kleine Büning, S. Martini, and M.M. Richter, editors, *CSL'92*, Springer LNCS, San Miniato, Italy, September 1992.

[6] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[7] M. Hanus. On extra variables in (equational) logic programming. In *Proc. Twelfth International Conference on Logic Programming*. MIT Press, 1995.

[8] J.R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

[9] Gérard Huet and Jean-Jacques Lévy. Computations in orthogonal rewriting systems, I. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 395–414. MIT Press, Cambridge, MA, 1991.

[10] John Wylie Lloyd. Combining functional and logic programming languages. In *Proceedings of the 1994 International Logic Programming Symposium, ILPS'94*, 1994.

[11] Hendrik C.R Lock. *The Implementation of Functional Logic Languages*. Oldenbourg Verlag, 1993.

[12] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. Technical report, Institut für Informatik, TU München, 1994.

[13] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1:497–536, 1991.

[14] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In C. Hogger D. Gabbay and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5. Oxford University Press. To appear.

[15] Tobias Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, 1991.

[16] Christian Prehofer. Decidable higher-order unification problems. In *Automated Deduction — CADE-12*. Springer LNAI 814, 1994.

[17] Christian Prehofer. Higher-order narrowing. In *Proc. Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, July 1994.

[18] Christian Prehofer. Higher-order narrowing with convergent systems. In *4th Int. Conf. Algebraic Methodology and Software Technology, AMAST '95*. Springer LNCS 936, July 1995.

[19] Christian Prehofer. *Solving Higher-order Equations: From Logic to Programming*. PhD thesis, TU München, 1995. Also appeared as Technical Report I9508.

[20] Zhenyu Qian. Higher-order equational logic programming. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, Portland, 1994.

[21] Yeh-heng Sheng. HIFUNLOG: Logic programming with higher-order relational functions. In David H.D. Warren and Peter Szeredi, editors, *Logic Programming*. MIT Press, 1990.