Higher-Order Functional-Logic Programming: A Systematic Development

Christian Prehofer
Institut für Informatik, Technische Universität München
80290 München, Germany
E-mail: prehofer@informatik.tu-muenchen.de

ABSTRACT

We develop an effective model for higher-order functional-logic programming by refining higher-order narrowing calculi. The refinements reduce the high degree of non-determinism in narrowing calculi, utilizing properties of functional(-logic) programs. These include convergent and left-linear rewrite rules. All refinements can be combined to a narrowing strategy which generalizes call-by-need as in functional programming. Furthermore, we consider conditional rewrite rules which are often convenient for programming applications.

1. Introduction

We present a systematic development of a calculus which integrates higher-order functional and logic programming, based on narrowing. Narrowing is a general method for solving equations modulo a set of rewrite rules. Functional-logic languages with a sound and complete operational semantics are mainly based on narrowing. For a survey on the topic we refer to [9].

In our higher-order equational logic we use a rewrite relation due to Nipkow [18], which computes on simply typed λ-terms modulo the conversions of λ-calculus. Higher-order rewriting allows for highly expressive rules, e.g. symbolic differentiation. The function $\mathit{diff}(F, X)$, defined by

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

computes the differential of a function F at a point X. With these rules, we can not only evaluate, as e.g.,

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

but also solve goals modulo these rules by narrowing. In contrast to rewriting, narrowing uses unification rules to instantiate free variables in order to find solutions to equational goals. We use directed equational goals of the form $s \rightarrow^? t$, where a substitution θ is a solution if $\theta s \xrightarrow{*} t$. Intuitively, the computation in such goals proceeds from left to right.

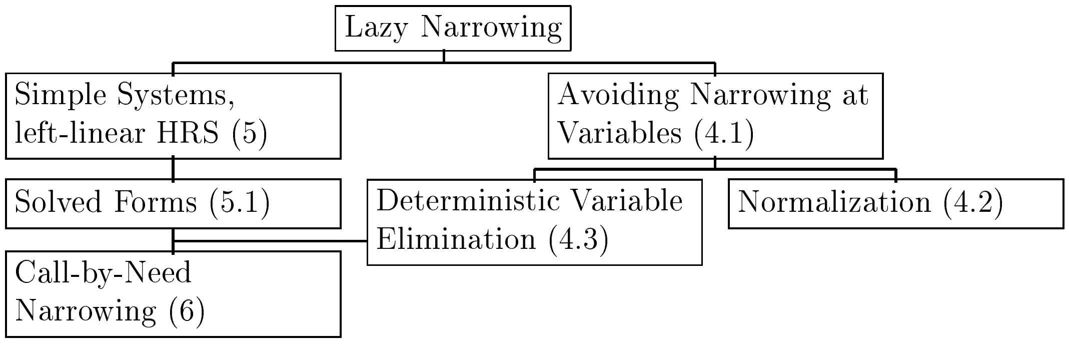
Lazy Narrowing
Simple Systems, left-linear HRS (5)
Avoiding Narrowing at Variables (4.1)
Solved Forms (5.1)
Deterministic Variable Elimination (4.3)
Normalization (4.2)
Call-by-Need Narrowing (6)

Figure 1: Dependencies of Lazy Narrowing Refinements

Since the naive versions of narrowing calculi are highly non-deterministic, many refinements have been developed. For an overview of first-order calculi we refer to [9]. The goal of this paper is to develop efficient higher-order calculi by refinement. This includes restricting some rules or imposing evaluation strategies, while still preserving completeness. As some of these refinements for lazy narrowing build upon others, we show their dependencies in Figure 1. Notice that all refinements can be combined in a straightforward way. It is one of our main contributions here to disassemble the many refinements used in the literature for optimizing narrowing calculi.

In the first, very general version of lazy narrowing in Section 3, we show the main ideas of lazy narrowing. The further refinements exploit well-examined and typical properties of rewrite systems and in particular programs: convergence and left-linearity, i.e. no variable occurs repeatedly in a left-hand side of a rule.

In Section 4.1, we focus on narrowing with a convergent rewrite system R. Unlike left-linearity, this setting permits powerful equational reasoning, as shown in Section 8.3. Like in the first-order case, this allows to restrict the most unconstrained case of narrowing: narrowing at variable positions. Further refinements include simplification via rewriting, as discussed in Section 4.2. This is desirable, as simplification is a deterministic operation. Another optimization, deterministic eager variable elimination, is examined in Section 4.3.

In Section 5, we focus on refinements of lazy narrowing for functional-logic programming, where the restriction to left-linear rules is both common and useful. We show that a certain class of equational goals, called Simple Systems, suffices. Furthermore, solved forms are much easier to detect than in the general case.

Combining the results for convergent systems with the properties of Simple Systems in Section 4.3 leads to an effective narrowing strategy, named Call-by-Need Narrowing. The basis for this strategy is a classification of the variables occurring in Simple Systems in Section 5.1. This allows to recognize and to delay intermediate goals, which are only solved when needed.

For both of the above settings, i.e. convergent and left-linear rules, we incorporated conditional rules. This common extension of term rewriting is useful in many applications. In Section 7 we introduce a class of conditional rules, called normal conditional rules. In a higher-order setting this class is sufficiently expressive for programming, although we do not permit extra variables on the right-hand sides of conditions, as discussed in Section 7.1.

This paper combines and summarizes several earlier papers [27, 29, 28]; more details and proofs can be found in [30].

2. Preliminaries

We briefly introduce simply typed λ-calculus (see e.g. [12]). 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 $t\updownarrow_\beta^\eta$, is the η-expanded form of the $\beta\eta$-normal form of t. It is well known [12] that $s =_{\alpha\beta\eta} t$ iff $s\updownarrow_\beta^\eta =_\alpha t\updownarrow_\beta^\eta$. 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 [3]) 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 [20]. Also, the unification of a linear pattern with a second-order term is decidable and finitary, if they are variable-disjoint [26]. 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** [23, 18] 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 of rewrite rules R if no rule from R applies and a substitution θ is **R-normalized** if all terms in the image of θ are in R-normal form. For other standard definitions of rewrite systems we refer to [4, 18].

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 in order 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. Lazy Narrowing

In this section, we introduce the central narrowing calculus which is used for functional-logic programming. Our setting for goal-directed lazy narrowing is as follows. We start with a goal $s \rightarrow^? t$, where a substitution θ is a solution if $\theta s \xrightarrow{*}^R t$. This goal may be simplified to smaller goals by the narrowing rules, which include the rules of higher-order unification.

For the rules of System LN, shown in Figure 2, we need some notation. Let $s \overset{?}{\leftrightarrow} t$ stand for one of $s \rightarrow^? t$ and $t \rightarrow^? s$. 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: $\{\overline{s \rightarrow^? t}\} \cup S \Rightarrow^\theta \{\overline{s_n \rightarrow^? t_n}\} \cup \theta S$ if $s \rightarrow^? t \Rightarrow^\theta \{\overline{s_n \rightarrow^? t_n}\}$. For a sequence $\Rightarrow^{\theta_1} \ldots \Rightarrow^{\theta_n}$ of LN 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.

System LN for lazy higher-order narrowing essentially consists of the rules for higher-order unification [34] plus the Lazy Narrowing rule. Observe that the first five rules in Figure 2 apply symmetrically as well, in contrast to the narrowing rules. For a first impression of lazy narrowing, we start with a few examples. Assuming the rules

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

we solve the goal $R(mary) \rightarrow^? art$ by

$$\begin{array}{lll} R(mary) \rightarrow^? art & \Rightarrow_{LN} & \text{Narrowing at Variable,} \\ & & R \mapsto \lambda x.father(R_1(x)) \\ R_1(mary) \rightarrow^? john, art \rightarrow^? art & \Rightarrow_{LN} & \text{Deletion or Decomposition} \\ R_1(mary) \rightarrow^? john & \Rightarrow_{LN} & \text{Narrowing at Variable,} \\ & & R_1 \mapsto \lambda x.father(R_2(x)) \\ R_2(mary) \rightarrow^? mary, john \rightarrow^? john & \Rightarrow_{LN} & \text{Projection} \\ mary \rightarrow^? mary, john \rightarrow^? john & \overset{*}{\Rightarrow}_{LN} & \text{Solved by Deletion} \end{array}$$

Thus we get the solution $R \mapsto \lambda x.father(father(x))$ by composing the partial bindings. Notice that the trivial solution $R \mapsto \lambda x.art$ is also possible here, but it is easy to avoid by further constraints as e.g. in the following example.

Another, slightly more involved example is the following. We use functional evaluation in this example for brevity.

$$\begin{array}{lll}
map(F,[mary,john]) \to^? [john,art] & \overset{*}{\Rightarrow}_{LN} & \text{Evaluation,} \\
 & & \text{Decomposition} \\
F(mary) \to^? john, F(john) \to^? art & \overset{*}{\Rightarrow}^{LN} & \text{Narrowing at Variable,} \\
 & & F \mapsto \lambda x.father(H(x)) \\
H(mary) \to^? mary, john \to^? john, & & \\
\quad father(H(john)) \to^? art & \Rightarrow_{LN} & Projection, H \mapsto \lambda x.x \\
mary \to^? mary, john \to^? john, & & \\
\quad father(john) \to^? art & &
\end{array}$$

The last goals are easily solved by evaluation and Deletion. This yields the solution $F \mapsto \lambda x.father(x)$. Observe how in the last examples Lazy Narrowing at Variable is used to compute solutions for functional variables. Although this rule is very powerful, it also has a high degree of non-determinism and will be restricted in later refinements.

There are two sources of non-determinism for such systems of transformations: which rules to apply and how to select the equations. Completeness fortunately does not depend on the goal selection, as each subgoal is independently solvable.

Definition 3.1 A (higher-order) narrowing calculus $\mathcal{N}$ is complete for some HRS R if the following holds: If $s \to^? t$ has solution θ, i.e. $\theta s \xrightarrow{*}{}^{R} \theta t$, then $\{s \to^? t\} \overset{*}{\Rightarrow}{}^{\delta}_{\mathcal{N}} F$ such that δ is more general, modulo the newly added variables, than θ and F is a set of flex-flex goals.

Theorem 3.2 *System LN is complete.*

4. Refinements Using the Determinism of Functional Languages

This section develops several refinements which exploit the determinism of convergent systems. For convergent systems it is sufficient to consider normalized solutions and particular reductions in the proofs, which allows to remove redundancies in the solutions.

4.1. Avoiding Lazy Narrowing at Variables

An essential refinement is to avoid narrowing at a variable occurring in a term of the form $X(\overline{x_n})$ for a R-normalized substitution θ with an HRS R. For a pattern $X(\overline{x_n})$, reducibility of a term $\theta X(\overline{x_n})$ implies that θ is not R-normalized, hence violating the assumption on θ. This result generalizes the first-order case, as for first-order terms narrowing at variable position is not needed. It is an important optimization, as narrowing at variable positions is highly unrestricted and thus creates large search

Deletion

$$t \to^{?} t \quad \Rightarrow \quad \{\}$$

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

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

Imitation

$$\begin{array}{lll} \lambda\overline{x_k}.F(\overline{t_n}) \overset{?}{\leftrightarrow} \lambda\overline{x_k}.f(\overline{t'_m}) & \Rightarrow^{\theta} & \{\overline{\lambda\overline{x_k}.H_m(\overline{\theta t_n}) \overset{?}{\leftrightarrow} \lambda\overline{x_k}.\theta t'_m}\} \\ & & \text{where } \theta = \{F \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\} \\ & & \text{with fresh variables } \overline{H_m} \end{array}$$

Projection

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

Lazy Narrowing with Decomposition

$$\begin{array}{lll} \lambda\overline{x_k}.f(\overline{t_n}) \to^{?} \lambda\overline{x_k}.t & \Rightarrow & \{\overline{\lambda\overline{x_k}.t_n \to^{?} \lambda\overline{x_k}.l_n},\ \lambda\overline{x_k}.r \to^{?} \lambda\overline{x_k}.t\} \\ & & \text{where } f(\overline{l_n}) \to r \text{ is an } \overline{x_k}\text{-lifted rule} \end{array}$$

Lazy Narrowing at Variable

$$\begin{array}{lll} \lambda\overline{x_k}.H(\overline{t_n}) \to^{?} \lambda\overline{x_k}.t & \Rightarrow^{\theta} & \{\overline{\lambda\overline{x_k}.H_m(\overline{\theta t_n}) \to^{?} \lambda\overline{x_k}.l_m},\ \lambda\overline{x_k}.r \to^{?} \lambda\overline{x_k}.\theta t\} \\ & & \text{where } f(\overline{l_m}) \to r \text{ is an } \overline{x_k}\text{-lifted rule,} \\ & & \text{and } \theta = \{H \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\} \\ & & \text{with fresh variables } \overline{H_m} \end{array}$$

Figure 2: System LN for Lazy Narrowing

spaces. We conjecture that in practice, as in higher-order logic programming [19], most terms are patterns and hence narrowing at variables is not needed very often.

To establish this result we need innermost reductions, which evaluate terms at inner redices first. For any solution there exists an innermost reduction, if R is convergent.

Definition 4.1 System **LNN** is defined as the restriction of system LN where Lazy Narrowing at Variable is not applied to goals of the form $\lambda\overline{x_n}.X(\overline{y_m}) \rightarrow^? t$ if $\lambda\overline{x_n}.X(\overline{y_m})$ is a higher-order pattern.

Theorem 4.2 *System LNN is complete for convergent R wrt R-normalized solutions.*

4.2. Simplification via Functional Evaluation

Simplification by normalization of goals is one of the earliest and one of the most important optimizations [8]. Its motivation is to prefer deterministic reduction over search within narrowing.

Observe that normalization coincides with deterministic evaluation in functional languages. Also, as shown below, deterministic operations are possible as soon as the left-hand side of a goal has been simplified to a term with a constructor at its root. For instance, with the rule $f(1) \rightarrow 1$, we can simplify a goal $f(1) \rightarrow^? g(Y)$ by $\{f(1) \rightarrow^? g(Y), \ldots\} \Rightarrow \{1 \rightarrow^? g(Y), \ldots\}$ and deterministically detect a failure.

For oriented goals, normalization is only complete for goals $s \rightarrow^? t$, where θt is in R-normal form for a solution θ. For instance, it suffices if t is a ground term in R-normal form. For most applications, this is no real restriction and corresponds to the intuitive understanding of directed goals.

Definition 4.3 Normalizing Lazy Narrowing is defined as the rules of LNN plus arbitrary simplification steps on the left-hand sides of goals.

Theorem 4.4 *Normalizing Lazy Narrowing is complete for convergent R wrt R-normalized solutions.*

4.3. Deterministic Eager Variable Elimination

Eager variable elimination is a particular strategy which has been examined for general equational unification. The idea is to apply the Elimination rule as a deterministic operation whenever possible. That is, when elimination applies to a goal, all other rule applications are not considered.

Eager variable elimination is of great practical value. Consider for instance the following example using the rules of Section 1:

$$\begin{array}{lll}
\{\lambda x.\mathit{diff}(\lambda y.1 * ln(F(y)), x) \rightarrow^? \lambda x.cos(x)/sin(x)\} & \overset{*}{\Rightarrow} & \text{Evaluation for } *, \mathit{diff} \\
\{\lambda x.\mathit{diff}(\lambda y.F(y), x)/F(x) \rightarrow^? \lambda x.cos(x)/sin(x)\} & \overset{*}{\Rightarrow} & \text{Decomposition} \\
\{\lambda x.\mathit{diff}(\lambda y.F(y), x) \rightarrow^? \lambda x.cos(x), & & \\
\ \lambda x.F(x) \rightarrow^? \lambda x.sin(x)\} & \overset{*}{\Rightarrow} & \text{Elimination}
\end{array}$$

At this point, we have to chose which goal to solve first. Since on the second Elimination applies, we prefer this and consider no further rules. It remains the goal

$$\{\lambda x.\mathit{diff}(\lambda y.\mathit{sin}(y), x) \rightarrow^? \lambda x.\mathit{cos}(x)\}$$

Unfortunately, it is an open problem of general (first-order) equational unification if eager variable elimination is still complete [33]. In our case, with oriented goals, we obtain more precise results by differentiating the orientation of the goal to be eliminated. As we consider oriented equations, we can distinguish two cases of variable elimination. For goals $X \rightarrow^? t$, elimination is deterministic as we can show that for a solution θ, θt is in R-normal form.

Theorem 4.5 *Lazy Narrowing with eager variable elimination on goals $X \rightarrow^? t$ where $X \notin \mathcal{FV}(t)$ is complete for convergent HRS R*

Interestingly, in [10] the elimination is purposely avoided in a programming language context, as it may copy terms whose evaluation can be expensive. This applies here as well to the remaining case $t \rightarrow^? X$, as shown later.

4.4. Failure Rules for Constructors

It is useful to add some refinements for constructors. First, decomposition on constructors, e.g. on $c(\ldots) \rightarrow^? c(\ldots)$ is deterministic. Correspondingly, goals of the form $c(\ldots) \rightarrow^? v(\ldots)$, where v is not a variable and $v \neq c$ are unsolvable, since evaluation proceeds from left to right.

5. Left-Linear Programs and Simple Systems

In this section we examine a particular class of goal systems, Simple Systems [27], which suffice for programming and have several interesting properties. We assume in the following an HRS with left-linear rules. A rule $l \rightarrow r \Leftarrow \overline{l_n \rightarrow r_n}$ is **left-linear**, if no free variable occurs repeatedly in l.

Definition 5.1 We write $s \rightarrow^? s' \ll t \rightarrow^? t'$, if $\mathcal{FV}(s') \cap \mathcal{FV}(t) \neq \{\}$.

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

Definition 5.2 A system of goals $\overline{G_n} = \overline{s_n \rightarrow^? t_n}$ 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 $\overline{t_n}$.

This class is closed under the rules of LN.

Theorem 5.3 *Assume a left-linear HRS R. If $\overline{G_n}$ is a Simple System, then applying LN with R preserves this property.*

The following results on solved forms from [27] will be crucial later.

Theorem 5.4 *A Simple System $S = \{X_1 \overset{?}{\leftrightarrow} t_1, \ldots, X_n \overset{?}{\leftrightarrow} t_n\}$ has a solution if all $\overline{X_n}$ are distinct.*

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

Corollary 5.5 *A Simple System of the form $\{\overline{t_n \rightarrow^? X_n}\}$ is solvable.*

In the second-order case unification never leads to divergence [27], extending results in [26] on unification with linear patterns:

Theorem 5.6 *Solving a second-order Simple System by the unification rules of LN, i.e. without the narrowing rules, terminates.*

5.1. 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 LN. 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 LN. We start with a goal $s \rightarrow^? t$ where initially the variables of interest are in s. This has to be updated for each LN step. If X is a variable of interest, and an LN 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 5.7 Assume a sequence of transformations $\{s \rightarrow^? t\} \overset{*}{\Rightarrow}{}^{\delta}_{LN} \{\overline{s_n \rightarrow^? t_n}\}$. 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:

Corollary 5.8 *Assume a left-linear HRS R and assume solving a Simple System $s \rightarrow^? t$ with system LN. 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, $X \rightarrow^? t$, the variable X is a variable of interest. Thus the elimination of X is desirable for computational reasons and is deterministic. 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.

$$t \rightarrow^? X,$$

elimination may not be deterministic. Hence 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 $f(X, Y) \rightarrow g(X, X)$:

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

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 $t_1{\downarrow_R} \rightarrow^? X$ and $t_2{\downarrow_R} \rightarrow^? Y$. The disadvantage is that eager evaluation may perform unnecessary evaluation steps.

Call-by-name is obtained by immediate eager variable elimination on $t_1 \rightarrow^? X$ and on $t_2 \rightarrow^? Y$. 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 $t_1 \rightarrow^? X$ and $t_2 \rightarrow^? Y$, thus avoiding copying. Then t_1 and t_2 are only evaluated when X or Y are needed for further computation.

In the sequel, 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 [1], but not to optimal or needed reduction [14].

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

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

In contrast to evaluation as in functional languages, solving the goals $t_1 \rightarrow^? X, t_2 \rightarrow^? Y$ 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 LN where goals of the form $t \rightarrow^? X$ are delayed.

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

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

is complete, or safe, in the sense that when only goals of the form $\overline{t_n \rightarrow^? X_n}$ remain, they are solvable by Corollary 5.5. Since the strategy is to delay such goals, this result is essential. (This may conflict with flex-flex pairs in some special cases, as discussed below.)

delays intermediate variables only. As shown in the last section, we can identify the variables to be delayed: a variable X in a goal $t \rightarrow^? X$ 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 $t \rightarrow^? X$ 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.

Completeness seems to follow easily from the results in last sections, but there is a technical problem with flex-flex pairs. Solved forms as in Corollary 5.5 do not extend to flex-flex pairs, as discussed in [30]. This seems however to occur rarely, and there

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

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 $\overline{H_m}$

Figure 3: Rule for System CLN for Conditional Lazy Narrowing

even exist classes of rewrite rules, which include functional programs, for which this problem does not occur.

7. Conditional Lazy Narrowing

In this section, we propose a class of conditional higher-order rewrite rules which is tailored for functional-logic programming languages. Then we introduce a system of transformations for this class of rules.

Definition 7.1 A **normal conditional HRS (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 $s \longrightarrow_{p,\theta}^{l\to r}$ 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 $\theta l_n \xleftrightarrow{*} \theta r_n$ is equivalent with $\theta l_n \xrightarrow{*} r_n$. The rules of System CLN for conditional narrowing, which differ from System LN, are shown in Figure 3.

The main ingredient for completeness of conditional narrowing is to assure that solutions for fresh variables in the conditions are normalized. This 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 7.2 *Conditional narrowing with CLN is complete for confluent and weakly normalizing NCHRS R.*

It is shown in [30] that all the results for unconditional rules can be extended for this class of conditional rules, assuming termination criteria.

7.1. Conditional Rules and Extra Variables

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 tuned notation for a let-construct is defined via

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

where the right side 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:

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

8. Examples

This section presents examples for higher-order functional-logic programming; more example can be found in [30]. Many examples use strict equality on first-order data types, which is also 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

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

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. For some applications full equality is useful and can be encoded via a rule $X = X \to true$, as used in Section 8.3.

8.1. Symbolic Differentiation

Using the rules for differentiation of Section 1, we can solve the following goal by call-by-need narrowing.

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

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

8.2. A Functional-Logic Parser

Top-down parsers belong to the classical examples for logic programming. The support for non-determinism in logic programming is the main ingredient for this application. On the other hand, functional parsers (see e.g. [24]) have other benefits, such as abstraction over parsing functions. We will integrate the best of both approaches in this example.

We model the following tiny grammar, which is similar to [24]: A grammar is described by the following terms. In addition to terminal symbols, e.g. a, b, c, we have the constructs $and(T,T')$, $or(T,T')$, and $rep(T)$. Their meaning is shown in the following table. For example, $and(t(a), or(t(b), t(c)))$ recognizes $[a, b]$ and $[a, c]$.

Construct	recognizes
t(a)	a, where a is a terminal symbol
$and(T,T')$	wv if T recognizes w and T' recognizes v.
$or(T,T')$	v if T or T' recognizes v,
$rep(T)$	$v_1 \dots v_n$ if T recognizes each v_i.

In our setting, each of these constructs is represented as a parsing function (of the same name). The main issue of this example is to show how to model non-deterministic constructs such as the parsing function for *or* with confluent rewrite rules. The solution is to add an extra argument, called prophecy, to *or*, which determines the choice. When invoking *or* with a free variable as prophecy, the desired effect is archived and in addition, the prophecies tell us which choice was made at each *or* construct.

In the following rules, T, T' represent parsing expressions and L is the input list of terminals. For uniformity we use prophecies for all parsing constructs, and not only for *or*. The constant symbols $pand, p1, p2, pt$ are used as prophecies. The function $t(x)$ is used to parse the terminal x.

$$\begin{array}{lcll}
t(X, pt, [Y|L]) & \rightarrow & L & \Leftarrow X =_s Y \\
and(T, T', pand(P1, P2), L) & \rightarrow & \mathsf{let}\ T(P1, L)\ \mathsf{in}\ \lambda l.T'(P2, l) & \\
or(T, T', por1(P), L) & \rightarrow & T(P, L) & \\
or(T, T', por2(P), L) & \rightarrow & T'(P, L) & \\
rep(T, pemp, L) & \rightarrow & L & \\
rep(T, prep(P, P1), L) & \rightarrow & \mathsf{let}\ T(P, L)\ \mathsf{in}\ \lambda l.T'(P2, l) &
\end{array}$$

The following example illustrates how parsing is performed:

$$and(t(a), or(t(b), t(c)), P, [a, b]) \rightarrow^? []$$

succeeds with

$$P \mapsto pand(pt, por1(pt)).$$

This solution for the prophecy can be seen as a parsing script showing how the word was parsed. Here, the first choice in the *or* construct was chosen.

As another example consider the goal

$$rep(or(t(b), t(c)), P, [b, c, b]) \rightarrow^? []$$

whose parsing function accepts words of b's and c's. The goal succeeds here with

$$P \mapsto prep(por1(t), prep(por2(t), prep(por1(t), pemp))).$$

In purely functional versions of this parser, the needed mechanism for non-determinism and search has to be coded by some means. Compared to first-order logic programming, we archive a higher level of abstraction and flexibility. For instance, the *rep* construct can be used with any parsing function (of the right type). In a first-order version, the constructs *and*, *or* etc would be represented as a data structure, and a function/predicate has to be written to interpret such constructs. Thus, when passing such a data structure, representing a parser, to some other function, this function has to know how to interpret the structure.

8.3. Program Transformation

The utility of higher-order unification for program transformations has been shown nicely by Huet and Lang [13] and has been developed further in [25, 7]. This example, taken from [5], applies unfold/fold program transformation and requires full instead of strict equality. We assume the following standard rules for lists

$$\begin{array}{rcl} map(F,[X|R]) & \rightarrow & [F(X)|map(F,R)] \\ foldl(G,[X|R]) & \rightarrow & G(X, foldl(G,R)) \end{array}$$

Now assume writing a function $g(F,L)$ by

$$g(F,L) \quad \rightarrow \quad foldl(\lambda x, y.plus(x,y), map(F,L))$$

that first maps F onto a list and then adds the elements via the function *plus*. This simple implementation for g is inefficient, since the list must be traversed twice. The goal is now to find an equivalent function definition that is more efficient. We can specify this with higher-order terms in a syntactic fashion by one simple equation:

$$\forall f, x, l.g(f,[x|l]) = B(f(x), g(f,l))$$

The variable B represents the body of the function to be computed and the first argument of B allows to use $f(x)$ in the body. The scheme on the right only allows recursing on l for g.

To solve this equation, we add a rule $X = X \rightarrow true$ and then apply narrowing, which yields the solution $\theta = \{B \mapsto \lambda fx, rec.plus(fx, rec)\}$ where

$$g(f,[x|l]) = \theta B(f(x), g(f,l)) = plus(f(x), g(f,l)).$$

This shows the more efficient definition of g. In this example, simplification can reduce the search space for narrowing drastically: it suffices to simplify the goal to

$$\lambda f, x, l.plus(f(x), foldl(plus, map(f,l))) = \lambda f, x, l.B(f(x), foldl(plus, map(f,l))),$$

where narrowing with the newly added rule $X = X \rightarrow true$ yields the two goals

$$\begin{array}{rcl} \lambda f, x, l.plus(f(x), foldl(plus, map(f,l))) & \rightarrow^? & \lambda f, x, l.X(f,x,l), \\ \lambda f, x, l.B(f(x), foldl(plus, map(f,l))) & \rightarrow^? & \lambda f, x, l.X(f,x,l). \end{array}$$

These can be solved by pure higher-order unification.

9. Conclusions and Related Work

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.

In contrast to many other approaches to higher-order functional-logic programming [6, 17, 32, 15, 16], we cover the full higher-order case and give completeness results. Similarly, other works on higher-order narrowing either examine more restricted cases [31, 2] or use an applicative first-order setting [22]. Compared to higher-order logic programming [21], higher-order programming as in functional languages is possible directly here.

A further interesting refinement for higher-order narrowing has been presented in [11]. To achieve optimality results, some further restrictions on rewrite rules are needed. Most of the examples we show however fit into this framework. It remains for further work to investigate the potential of this higher-order setting and in particular to work towards efficient implementations.

Acknowledgments This research was supported by the DFG under grant Br 887, *Deduktive Programmentwicklung* and by ESPRIT WG 6028, *CCL*.

References

[1] 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.

[2] 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.

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

[4] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320. Elsevier, 1990.

[5] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, Wokingham, 1988.

[6] 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.

[7] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In *Fifth International Logic Programming Conference*, pages 942–959, Seattle, Washington, August 1988. MIT Press.

[8] 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*, pages 1–23. Springer LNCS 631, 1992.

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

[10] M. Hanus. Lazy unification with simplification. In *Proc. 5th European Symposium on Programming*, pages 272–286. Springer LNCS 788, 1994.

[11] M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. In *Proc. Seventh International Conference on Rewriting Techniques and Applications (RTA'96)*. Springer LNCS 1103, 1996.

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

[13] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[14] 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.

[15] H. Kuchen and J. Anastasiadis. Higher Order Babel — language and implementation. In *Proceedings of Extensions of Logic Programming.* Springer LNCS 1050, 1996.

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

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

[18] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science.* To appear.

[19] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In Dale Miller, editor, *Proceedings of the Workshop on the Lambda Prolog Programming Language*, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania.

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

[21] 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.

[22] K. Nakahara, A. Middeldorp, and T. Ida. A complete narrowing calculus for higher-order functional logic programming. In *Proceedings of Seventh International Conference on Programming Language Implementation and Logic Programming 95 (PLILP'95), Springer LNCS 982*, pages 97–114, 1995.

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

[24] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[25] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proc. SIGPLAN '88 Symp. Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

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

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

[28] Christian Prehofer. A Call-by-Need Strategy for Higher-Order Functional-Logic Programming. In J. Lloyd, editor, *Logic Programming. Proc. of the 1995 International Symposium*, pages 147–161. MIT Press, 1995.

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

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

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

[32] 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.

[33] Wayne Snyder. *A Proof Theory for General Unification.* Birkäuser, Boston, 1991.

[34] Wayne Snyder and Jean Gallier. Higher-order unification revisited: Complete sets of transformations. *J. Symbolic Computation*, 8:101–140, 1989.