In *Proc. Seventh Intern. Conference on Rewriting Techniques and Applications* (RTA'96), New Brunswick (USA), Springer LNCS 1103, pp. 138–152, 1996

Higher-Order Narrowing with Definitional Trees

Michael Hanus[1] and Christian Prehofer[2]

[1] Informatik II, RWTH Aachen, D-52056 Aachen, Germany
hanus@informatik.rwth-aachen.de
[2] Fakultät für Informatik, TU München, D-80290 München, Germany
prehofer@informatik.tu-muenchen.de

Abstract. Functional logic languages with a sound and complete operational semantics are mainly based on narrowing. Due to the huge search space of simple narrowing, steadily improved narrowing strategies have been developed in the past. Needed narrowing is currently the best narrowing strategy for first-order functional logic programs due to its optimality properties w.r.t. the length of derivations and the number of computed solutions. In this paper, we extend the needed narrowing strategy to higher-order functions and λ-terms as data structures. By the use of definitional trees, our strategy computes only incomparable solutions. Thus, it is the first calculus for higher-order functional logic programming which provides for such an optimality result. Since we allow higher-order logical variables denoting λ-terms, applications go beyond current functional and logic programming languages.

1 Introduction

Functional logic languages [7] with a sound and complete operational semantics are mainly based on narrowing. Narrowing, originally introduced in automated theorem proving [20], is used to *solve* goals by finding appropriate values for variables occurring in arguments of functions. A *narrowing step* instantiates 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 less-or-equal predicate on natural numbers which are represented by terms built from 0 and s:

$$0 \le X \to true$$
$$s(X) \le 0 \to false$$
$$s(X) \le s(Y) \to X \le Y$$

To solve the goal $s(X) \le Y$, we perform a first narrowing step by instantiating Y to $s(Y_1)$ and applying the third rule, and a second narrowing step by instantiating X to 0 and applying the first rule:

$$s(X) \le Y \quad \leadsto_{\{Y \mapsto s(Y_1)\}} \quad X \le Y_1 \quad \leadsto_{\{X \mapsto 0\}} \quad true$$

Since the goal is reduced to $true$, the computed solution is $\{X \mapsto 0, Y \mapsto s(Y_1)\}$.

Due to the huge search space of simple narrowing, steadily improved narrowing strategies have been developed in the past. *Needed narrowing* [2] is based on the idea to evaluate only subterms which are *needed* in order to compute some result. For instance, in a goal $t_1 \leq t_2$, it is always necessary to evaluate t_1 (to some head normal form) since all three rules in Example 1 have a non-variable first argument. On the other hand, the evaluation of t_2 is only needed if t_1 is of the form $s(\cdots)$. Thus, if t_1 is a free variable, needed narrowing instantiates it to a constructor, here 0 or s. Depending on this instantiation, either the first rule is applied or the second argument t_2 is evaluated. Needed narrowing is the currently best narrowing strategy for first-order functional logic programs due to its optimality properties w.r.t. the length of derivations and the number of computed solutions [2]. Moreover, it can be efficiently implemented by pattern-matching and unification due to its local computation of a narrowing step (see, e.g., [8]).

In this paper, we extend the needed narrowing strategy to higher-order functions and λ-terms as data structures. We introduce a class of higher-order inductively sequential rewrite rules which can be defined via definitional trees. Although this class is a restriction of general higher-order rewrite systems, it covers higher-order functional languages. As higher-order rewrite steps can be expensive in general, we show that finding a redex with inductively sequential rules can be performed as in the first-order case.

Since our narrowing calculus LNT is oriented towards previous work on higher-order narrowing [19], we show in the first part that LNT coincides with needed narrowing in the first-order case. For the higher-order case, we show soundness and completeness with respect to higher-order needed reductions, which we define via definitional trees. Furthermore, we show that the calculus is optimal w.r.t. the solutions computed, i.e., no solution is produced twice. Optimality of higher-order reductions is subject of current research. It is however shown that higher-order needed reductions are in fact needed for reduction to a constructor normal form.

This strategy is the first calculus for higher-order functional logic programming which provides for optimality results. Moreover, it falls back to the optimal needed narrowing strategy if the higher-order features are not used, i.e., our calculus is a conservative extension of an optimal first-order narrowing calculus. Since we allow higher-order logical variables denoting λ-terms, applications go beyond current functional and logic programming languages. In general, our calculus can compute solutions for variables of functional type. Although this is very powerful, we show that the incurring higher-order unification can sometimes be avoided by techniques similar to [4]. Due to lack of space, some details and the proofs are omitted. They can be found in [9].

2 Preliminaries

We briefly introduce the simply typed λ-calculus (see e.g. [10]). We assume the following **variable conventions**:

- F, G, H, P, 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 $\mathcal{T}$ for the simply typed λ-terms is generated by a set $\mathcal{T}_0$ of base types (e.g., int, bool) and the function type constructor $\rightarrow$. The syntax for **λ-terms** is given by

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

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. **Substitutions** are finite mappings from variables to terms, denoted by $\{\overline{X_n \mapsto t_n}\}$, and extend homomorphically from variables to terms. Free and bound variables of a term t will be denoted as $\mathcal{FV}(t)$ and $\mathcal{BV}(t)$, respectively. A term t is **ground** if $\mathcal{FV}(t) = \{\}$. The **conversions in λ-calculus** are defined as:

- **α-conversion:** $\lambda x.t =_\alpha \lambda y.(\{x \mapsto y\}t)$,
- **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 [14] of a term t, denoted by $t\updownarrow_\beta^\eta$, is the η-expanded form of the β-normal form of t. It is well known [10] 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 [5]) 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 $\alpha \in W$. The restriction of a substitution to a set of variables W is defined as $\theta_{|W}\alpha = \theta\alpha$ if $\alpha \in W$ and $\theta_{|W}\alpha = \alpha$ otherwise. 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 [12]. Examples are $\lambda x, y.F(x, y)$ and $\lambda x.f(G(\lambda z.x(z)))$.

A **rewrite rule** [14] is a pair $l \rightarrow r$ such that l is a higher-order 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

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

For a rewrite step we often omit some of the parameters $l \to r, p$ and θ. It is a standard assumption in functional logic programming that constant symbols are divided into free **constructor symbols** and defined symbols. A symbol f is called a **defined symbol** or **operation**, if a rule $f(\cdots) \to t$ exists. A **constructor term** is a term without defined symbols. Constructor symbols and constructor terms are denoted by c and d. A term $f(\overline{t_n})$ is called **operation-rooted** (respectively **constructor-rooted**) if f is a defined symbol (respectively constructor). A **higher-order rewrite system** (**HRS**) $\mathcal{R}$ is a set of rewrite rules. A term is in **$\mathcal{R}$-normal form** if no rule from $\mathcal{R}$ applies and a substitution θ is **$\mathcal{R}$-normalized** if all terms in the image of θ are in $\mathcal{R}$-normal form.

By applying rewrite steps, we can compute the *value* of a functional expression. However, in the presence of free variables, we have to compute values for these free variables such that the instantiated expression is reducible. This is the motivation for narrowing which will be precisely defined in the following sections. Narrowing is intended to *solve* goals, where a **goal** is an expression of Boolean type that should be reduced to the constant *true*. This is general enough to cover the equation solving capabilities of current functional logic languages with a lazy operational semantics, like BABEL [13] or K-LEAF [6], since the strict equality $\approx$[1] can be defined as a binary operation by a set of orthogonal rewrite rules (see [2, 6, 13] for more details about strict equality). An important consequence of this restriction on goals is the fact that during the successful rewriting of a goal the topmost symbol is always an operation or the constant *true*. This property will be used to simplify the narrowing calculus.

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, $\{G \mapsto G'(x)\}$ is an x-lifter of $g(G)$ away from any W not containing G'.

3 First-Order Definitional Trees

Definitional trees are introduced in [1] to define efficient normalization strategies for (first-order) term rewriting. The idea is to represent all rules for a defined

[1] The **strict equality** $t \approx t'$ holds if t and t' are reducible to the same ground constructor term. Note that normal forms may not exist in general due to non-terminating rewrite rules.

symbol in a tree and to control the selection of the next redex by this tree. This technique is extended to narrowing in [2]. We will extend definitional trees to the higher-order case in order to obtain a similar strategy for higher-order narrowing. To state a clear relationship between the first-order and the higher-order case, we review the first-order case in this section and present the needed narrowing calculus in a new form. Thus, we assume in this section that all terms are first-order, i.e., λ-abstractions and functional variables do not occur.

Traditionally [7], a term t is **narrowed** into a term t' if there exist a non-variable position p in t (i.e., $t|_p$ is not a free variable), a variant $l \to r$ of a rewrite rule with $\mathcal{FV}(t) \cap \mathcal{FV}(l \to r) = \{\}$ and a most general unifier σ of $t|_p$ and l such that $t = \sigma(t[r]_p)$. In this case we write $t \leadsto_\sigma t'$. We write $t_0 \leadsto^*_\sigma t_n$ if there is a narrowing derivation $t_0 \leadsto_{\sigma_1} t_1 \leadsto_{\sigma_2} \cdots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \cdots \sigma_2\sigma_1$. In order to compute all solutions by narrowing, we have to apply all rules at all non-variable subterms in parallel. Since this simple method leads to a huge and often infinite search space, many improvements have been proposed in the past (see [7] for a survey). A **narrowing strategy** determines the position where the next narrowing step should be applied. As shown in [2], an optimal narrowing strategy can be obtained by dropping the requirement for most general unifiers and controlling the instantiation of variables and selection of narrowing positions by a data structure, called definitional tree. $\mathcal{T}$ is a **definitional tree** with pattern π iff its depth is finite and one of the following cases holds:

$\mathcal{T} = rule(l \to r)$, where $l \to r$ is a variant of a rule in $\mathcal{R}$ such that $l = \pi$.
$\mathcal{T} = branch(\pi, o, \overline{\mathcal{T}_k})$, where o is an occurrence of a variable in π, $\overline{c_k}$ are different constructors of the type of $\pi|_o$ $(k > 0)$, and, for $i = 1, \ldots, k$, $\mathcal{T}_i$ is a definitional tree with pattern $\pi[c_i(\overline{X_{n_i}})]_o$, where n_i is the arity of c_i and $\overline{X_{n_i}}$ are new distinct variables.

A **definitional tree** of an n-ary function f is a definitional tree $\mathcal{T}$ with pattern $f(\overline{X_n})$, where $\overline{X_n}$ are distinct variables, such that for each rule $l \to r$ with $l = f(\overline{t_n})$ there is a node $rule(l' \to r')$ in $\mathcal{T}$ with l variant of l'.[2] For instance, the rules in Example 1 can be represented by the following definitional tree:

$$\begin{array}{l} branch(X \leq Y, 1, rule(\mathbf{0} \leq Y \to true), \\ \qquad\qquad\qquad\qquad branch(\boldsymbol{s}(X') \leq Y, 2, rule(s(X') \leq \mathbf{0} \to false), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad rule(s(X') \leq \boldsymbol{s}(Y') \to X' \leq Y'))) \end{array}$$

A definitional tree starts always with the most general pattern for a defined symbol and branches on the instantiation of a variable to constructor-headed terms, here 0 and $s(X')$. It is essential that each rewrite rule occurs only once as a leaf of the tree. Thus, when evaluating the arguments of a term $f(\overline{t_n})$ to constructor terms, the tree can be incrementally traversed to find the matching rule.

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 the rewrite system $\mathcal{R}$. The term rewriting system $\mathcal{R}$ is called inductively sequential if each function defined by $\mathcal{R}$ is inductively sequential.

[2] This corresponds to Antoy's notion [1] except that we ignore *exempt* nodes.

A definitional tree defines a strategy to apply narrowing steps.[3] To narrow a term t, we consider the definitional tree $\mathcal{T}$ of the outermost function symbol of t (note that, by our restriction on goals, the outermost symbol is always a Boolean function). If $\mathcal{T} = rule(l \to r)$, we apply the rule $l \to r$ to t. If $\mathcal{T} = branch(\pi, o, \overline{\mathcal{T}_k})$, we consider the subterm $t|_o$. 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 $t|_o$. 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. If $t|_o$ is a variable, we non-deterministically select a subtree $\mathcal{T}_j$, instantiate $t|_o$ to the constructor of the pattern of $\mathcal{T}_j$ at position o, and narrow this instance of t with $\mathcal{T}_j$. This strategy is called **needed narrowing** [2] and is the currently best narrowing strategy due to its optimality w.r.t. the length of derivations (if terms are shared) and the number of computed solutions.

In order to extend this strategy to higher-order functions, another representation is required since it is shown in [17] that the direct application of narrowing steps to inner subterms should be avoided in the presence of λ-bound variables. For this purpose we transform the needed narrowing calculus into a lazy narrowing calculus in the spirit of Martelli/Montanari's inference rules. In a first step, we integrate the definitional trees into the rewrite rules by extending the language of terms and providing *case* constructs to express the concrete narrowing strategy. A **case expression** has the form

$$case\ X\ of\ c_1(\overline{X_{n_1}}) : \mathcal{X}_1, \ldots, c_k(\overline{X_{n_k}}) : \mathcal{X}_k$$

where X is a variable, $c_1, \ldots, c_k$ are different constructors of the type of X, and $\mathcal{X}_1, \ldots, \mathcal{X}_k$ are terms possibly containing case expressions. Using such case expressions, each inductively sequential function symbol can be defined by exactly one rewrite rule. For instance, the rules for the function $\leq$ defined in Example 1 are represented by the following rule:

$$X \leq Y \ \to\ case\ X\ of\ 0 : true,\ s(X_1) : (case\ Y\ of\ 0 : false,\ s(Y_1) : X_1 \leq Y_1)$$

To be more precise, we translate a definitional tree $\mathcal{T}$ into a term with case expressions by the use of the function $dtc(\mathcal{T})$ which is defined as follows:

$$\begin{aligned} dtc(rule(l \to r)) &= r \\ dtc(branch(\pi, o, \overline{\mathcal{T}_k})) &= case\ \pi|_o\ of\ \pi_1|_o : dtc(\mathcal{T}_1), \ldots, \pi_k|_o : dtc(\mathcal{T}_k) \\ &\quad \text{where } \pi_i \text{ is the pattern of } \mathcal{T}_i \end{aligned}$$

If $\mathcal{T}$ is the definitional tree with pattern $f(\overline{X_n})$ of the n-ary function f, then $f(\overline{X_n}) \to dtc(\mathcal{T})$ is the new rewrite rule for f. A case expression $case\ X\ of\ \overline{p_n : \mathcal{X}_n}$ can be considered as a function with arity $2n + 1$ where the semantics is defined by the following n rewrite rules:[4]

$$case\ p_i\ of\ \overline{p_n : X_n} \to X_i \quad (i = 1, \ldots, n)$$

[3] Due to lack of space, we omit a precise definition which can be found in [2].

[4] To be more precise, different *case* functions are needed for case expressions with different patterns, i.e., the case functions should be indexed by the case patterns. However, for the sake of readability, we do not write these indices and allow the overloading of the *case* function symbols.

Bind

$$e \to^? Z, G \Rightarrow^\sigma \sigma(G)$$

if e is not a case term and $\sigma = \{Z \mapsto e\}$

Case Select

$$\mathit{case}\ c(\overline{t_n})\ \mathit{of}\ \overline{p_k : \mathcal{X}_k} \to^? Z, G \Rightarrow^\sigma \sigma(\mathcal{X}_i) \to^? Z,\ G$$

where $p_i = c(\overline{X_n})$ and $\sigma = \{\overline{X_n \mapsto t_n}\}$

Case Guess

$$\mathit{case}\ X\ \mathit{of}\ \overline{p_k : \mathcal{X}_k} \to^? Z, G \Rightarrow^\sigma \sigma(\mathcal{X}_i) \to^? Z,\ \sigma(G)$$

where $\sigma = \{X \mapsto p_i\}$

Case Eval

$$\mathit{case}\ f(\overline{t_n})\ \mathit{of}\ \overline{p_k : \mathcal{X}_k} \to^? Z, G \Rightarrow^\sigma \sigma(\mathcal{X}) \to^? X,\ \mathit{case}\ X\ \mathit{of}\ \overline{p_k : \mathcal{X}_k} \to^? Z, G$$

if $f(\overline{X_n}) \to \mathcal{X} \in \mathcal{R}'$ is a rule with fresh variables, $\sigma = \{\overline{X_n \mapsto t_n}\}$, and X is a fresh variable

Fig. 1. Calculus LNT for lazy narrowing with definitional trees in the first-order case

In the following, we denote by $\mathcal{R}$ an inductively sequential rewrite system, by $\mathcal{R}'$ its translated version containing exactly one rewrite rule for each function defined by $\mathcal{R}$, and by $\mathcal{R}_c$ the additional *case* rewrite rules. The following theorem states that needed narrowing w.r.t. $\mathcal{R}$ and leftmost-outermost narrowing w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$ are equivalent, where **leftmost-outermost** means that the selected subterm is the leftmost-outermost one among all possible narrowing positions.[5]

Theorem 1. *Let t be a term with a Boolean function at the top. For each needed narrowing derivation $t \leadsto^*_\sigma \mathit{true}$ w.r.t. $\mathcal{R}$ there exists a leftmost-outermost narrowing derivation $t \leadsto^*_{\sigma'} \mathit{true}$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$ with $\sigma =_{\mathcal{FV}(t)} \sigma'$, and vice versa.*

As mentioned above, in the higher-order case we need a narrowing calculus which always applies narrowing steps to the outermost function symbol which is often different from the leftmost-outermost narrowing position. For this purpose, we transform a leftmost-outermost narrowing derivation w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$ into a derivation on a **goal system** G (a sequence of goals of the form $t \to^? X$) where narrowing rules are only applied to the outermost function symbol of the leftmost goal. This is the purpose of the inference system LNT shown in Figure 1. The **Bind** rule propagates a term to the subsequent case expression. The **Case** rules correspond to the case distinction in the definition of needed narrowing, where the narrowing of a function is integrated in the **Case Eval** rule. Note that the only possible non-determinism during computation with these inference rules is in the **Case Guess** rule. Since we are interested in solving goals by reduction to *true*, we assume that the **initial goal** has always the form $\mathit{case}\ t\ \mathit{of}\ \mathit{true} : \mathit{true} \to^? T$. We use this representation in order to provide a calculus with few inference rules. Note that $T \mapsto \mathit{true}$ if such a goal can be reduced to the empty goal system.

[5] A position p is **leftmost-outermost** in a set P of positions if there is no $p' \in P$ with p' prefix of p, or $p' = q \cdot i \cdot q'$ and $p = q \cdot j \cdot q''$ and $i < j$.

Theorem 2. *Let t be a term with a Boolean function at the top and X a fresh variable. For each leftmost-outermost narrowing derivation $t \leadsto_{\sigma}^{*} true$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$ there exists a LNT-derivation $case\ t\ of\ true : true \to^{?} X \overset{*}{\Rightarrow}{}^{\sigma'} true \to^{?} X$ w.r.t. $\mathcal{R}'$ such that $\sigma' =_{\mathcal{FV}(t)} \sigma$, and vice versa.*

Theorems 1 and 2 imply the equivalence of needed narrowing and the calculus LNT. Since we will extend LNT to higher-order functions in the next section, the results in this section show that our higher-order calculus is a conservative extension of an optimal first-order narrowing strategy.

4 Higher-Order Definitional Trees

In the following we extend first-order definitional trees to the higher-order case. To generalize from the first-order case, it is useful to recall the main ideas: When evaluating the arguments of a term $f(\overline{t_n})$ to constructor terms, the definitional tree can be incrementally traversed to find the (single) matching rule. It is essential that each branching depends on only one subterm (or argument to the function) and that for each rigid term (non-variable headed), a single branch can be chosen. For this purpose, we need further restrictions in the higher-order case, where we employ λ-terms as data structure, e.g., higher-order terms with bound variables in the left-hand sides. For instance, we permit the rules

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

where $\mathit{diff}(F, X)$ computes the differential of F at X.

A **shallow pattern** is a linear term of the form $\lambda\overline{x_n}.v(\overline{H_m(\overline{x_n})})$. We will use shallow patterns for branching in trees. In contrast to the first-order case, v can also be a bound variable.

Definition 3. $\mathcal{T}$ is a **higher-order definitional tree** **(***hdt***)** iff its depth is finite and one of the following cases holds:

- $\mathcal{T} = p_f : case\ X\ of\ \overline{\mathcal{T}_n}$
- $\mathcal{T} = p_f : rhs$,

where p_f are shallow patterns with fresh variables, X is a free variable and $\overline{\mathcal{T}_n}$ are *hdt*s in the first case, and rhs is a term (representing the right-hand side of a rule). Moreover, all shallow patterns of the *hdts* $\overline{\mathcal{T}_n}$ must be pairwise non-unifiable.

We write *hdt*s as $p_f : \mathcal{X}$, where $\mathcal{X}$ stands for a case expression or a term. To simplify technicalities, rewrite rules $f(\overline{X_n}) \to \mathcal{X}$ are identified with the *hdt* $f(\overline{X_n}) : \mathcal{X}$. With this latter form of a rule, we can relate rules to the usual notation as follows. The **selector** of a tree $\mathcal{T}$ of the form $\mathcal{T} = p_f : \mathcal{X}$ is defined as $sel(\mathcal{T}) = p_f$. For a node $\mathcal{T}'$ in a tree $\mathcal{T}$, the constraints in the case expressions on the path to it determine a term, which is recursively defined by the pattern function $pat_{\mathcal{T}}(\mathcal{T}')$:

$$pat_{\mathcal{T}}(\mathcal{T}') = \begin{cases} sel(\mathcal{T}') & \text{if } \mathcal{T} = \mathcal{T}' \text{ (i.e., } \mathcal{T}' \text{ is the root)} \\ \{X \mapsto sel(\mathcal{T}')\} pat_{\mathcal{T}}(\mathcal{T}'') & \text{if } \mathcal{T}' \text{ has parent } \mathcal{T}'' = p_f : case\ X\ of\ \overline{\mathcal{T}_n} \end{cases}$$

Each branch variable must belong to the pattern of this node, i.e., for each node $\mathcal{T}' = p_f : case\ X\ of\ \overline{\mathcal{T}_n}$ in a tree $\mathcal{T}$, X is a free variable of $pat_{\mathcal{T}}(\mathcal{T}')$. Furthermore, each leaf $\mathcal{T}' = p : rhs$ of a *hdt* $\mathcal{T}$ is required to correspond to a rewrite rule $l \to r$, i.e., $pat_{\mathcal{T}}(\mathcal{T}') \to rhs$ is a variant of $l \to r$. $\mathcal{T}$ is called ***hdt* of a function** f if for all rewrite rules of f there is exactly one corresponding leaf in $\mathcal{T}$.

As in the first-order case, rewrite rules must be **constructor based.** This means that in a *hdt* only the outermost pattern has a defined symbol. An HRS, for all of which defined symbols *hdt*s exits, is called **inductively sequential**.

For instance, the rules for *diff* above have the *hdt*

$$\begin{aligned} diff(F,X) \to case\ F\ of\ & \lambda y.y && : 1, \\ & \lambda y.sin(F'(y)) && : cos(F'(X)) * diff(\lambda y.F'(y), X), \\ & \lambda y.ln(F'(y)) && : diff(\lambda y.F'(y), X)/F'(X) \end{aligned}$$

Note that free variables in left-hand sides must have all bound variables of the current scope as arguments. Such terms are called **fully extended**. This important restriction, which also occurs in [16], allows to find redices as in the first-order case, and furthermore simplifies narrowing. For instance, Flex-Flex pairs do not arise here, in contrast to the full higher-order case [18, 19]. Consider an example for some non-overlapping rewrite rules which do not have a *hdt*:

$$\begin{aligned} f(\lambda x.c(x)) &\to a \\ f(\lambda x.H) &\to b \end{aligned}$$

The problem is that for rewriting a term with these rules the full term must be scanned. For example, if the argument to f is the rigid term $\lambda x.c(G(t))$, it is not possible to commit to one of the rules (or branches of a tree) before checking if the bound variable x occurs inside t. In general, this may lead to an unexpected complexity w.r.t. the term size for evaluation via rewriting.

We define the $\overline{x_k}$-lifting of *hdt*s by schematically applying the $\overline{x_k}$-lifter to all terms in the tree, i.e., to all patterns, right-hand sides, and free variables in cases.

5 Narrowing with Higher-Order Definitional Trees

In the higher-order case, the rules of LNT of Section 3 must be extended to account for several new cases. Compared to the first-order case, we need to maintain binding environments and higher-order free variables, possibly with arguments, which are handled by higher-order unification. For this purpose, the Imitation, the Function Guess and the Projection rules have been added in Figure 2. These three new rules, to which we refer as the Guess Rules, are the only ones to compute substitutions for the variables in the case constructs. The Case Guess rule of the first-order case can be retained by applying Imitation plus Case Select. The Imitation and Projection rules are taken from higher-order unification and compute a partial binding for some variable. The Function Guess rule covers the case of non-constructor solutions, which may occur for higher-order variables. It thus enables the synthesis of functions from existing ones. Note that the selection of a binding in this rule is only restricted by the types occurring. For all rules, we assume that newly introduced variables are fresh, as in the first-order case.

Bind		
$e \to^? Z, G$	$\Rightarrow^{\{\}}$	$\sigma(G)$ where $\sigma = \{Z \mapsto e\}$ and e is not a case term
Case Select		
$\lambda\overline{x_k}.case\ \lambda\overline{y_l}.v(\overline{t_m})\ of$ $\overline{p_n : \mathcal{X}_n} \to^? Z, G$	$\Rightarrow^{\{\}}$	$\lambda\overline{x_k}.\sigma(\mathcal{X}_i) \to^? Z, G$ if $p_i = \lambda\overline{y_l}.v(\overline{X_m(\overline{x_k},\overline{y_l})})$ and $\sigma = \{\overline{X_m \mapsto \lambda\overline{x_k},\overline{y_l}.t_m}\}$
Imitation		
$\lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{t_m})\ of$ $\overline{p_n : \mathcal{X}_n} \to^? Z, G$	$\Rightarrow^{\sigma}$	$\sigma(\lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{t_m})\ of\ \overline{p_n : \mathcal{X}_n} \to^? Z, G)$ if $p_i = \lambda\overline{y_l}.c(\overline{X_o(\overline{x_k},\overline{y_l})})$ and $\sigma = \{X \mapsto \lambda\overline{x_m}.c(\overline{H_o(\overline{x_m})})\}$
Function Guess		
$\lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{t_m})\ of$ $\overline{p_n : \mathcal{X}_n} \to^? Z, G$	$\Rightarrow^{\sigma}$	$\sigma(\lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{t_m})\ of\ \overline{p_n : \mathcal{X}_n} \to^? Z, G)$ if $\lambda\overline{x_k},\overline{y_l}.X(\overline{t_m})$ is not a higher-order pattern, $\sigma = \{X \mapsto \lambda\overline{x_m}.f(\overline{H_o(\overline{x_m})})\}$, and f is a defined function
Projection		
$\lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{t_m})\ of$ $\overline{p_n : \mathcal{X}_n} \to^? Z, G$	$\Rightarrow^{\sigma}$	$\sigma(\lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{t_m})\ of\ \overline{p_n : \mathcal{X}_n} \to^? Z, G)$ where $\sigma = \{X \mapsto \lambda\overline{x_m}.x_i(\overline{H_o(\overline{x_m})})\}$
Case Eval		
$\lambda\overline{x_k}.case\ \lambda\overline{y_l}.f(\overline{t_m})\ of$ $\overline{p_n : \mathcal{X}_n} \to^? Z, G$	$\Rightarrow^{\{\}}$	$\lambda\overline{x_k},\overline{y_l}.\sigma(\mathcal{X}) \to^? X,$ $\lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{x_k},\overline{y_l})\ of\ \overline{p_n : \mathcal{X}_n} \to^? Z, G$ where $\sigma = \{\overline{X_m \mapsto \lambda\overline{x_k},\overline{y_l}.t_m}\}$, and $f(\overline{X_m(\overline{x_k},\overline{y_l})}) \to \mathcal{X}$ is a $\overline{x_k},\overline{y_l}$-lifted rule

Fig. 2. System LNT for needed narrowing in the higher-order case

Notice that for goals where only higher-order patterns occur, there is no choice between Projection and Imitation and furthermore Function Guess does not apply. This special case is refined later in Section 8.

For a sequence $\Rightarrow^{\theta_1} \cdots \Rightarrow^{\theta_n}$ of LNT steps, we write $\stackrel{*}{\Rightarrow}^{\theta}$, where $\theta = \theta_n \cdots \theta_1$. In contrast to the calculus in Section 3 not all substitutions are recorded for $\stackrel{*}{\Rightarrow}$; only the ones produced by guessing are needed for the technical treatment. Informally, all other substitutions only concern intermediate (or auxiliary) variables similar to [18].

As in the first-order case, we consider only reductions to the dedicated constant $true$. This is general enough to cover reductions to a term without defined symbols c, since a reduction $t \stackrel{*}{\longrightarrow} c$ can be modeled by $f(t) \stackrel{*}{\longrightarrow} true$ with the additional rule $f(c) \to true$ and a new symbol f. Hence we assume that solving a goal $t \to^? true$ is initiated with the **initial goal** $I(t) = case\ t\ of\ true : true \to^? X$.

As an example, consider the goal $\lambda x.diff(\lambda y.sin(F(x,y)),x) \to^? \lambda x.cos(x)$ w.r.t. the rules for *diff* and the *hdt* for the function $*$:

$$X * Y \to case\ Y\ of\ 1 : X, s(Y') : X + X * Y'$$

To solve the above goal, we simply add the rule $f(\lambda x.cos(x)) \to true$ to solve the following goal. Since each computation step only affects the two leftmost goals, we often omit the others.

$case\ f(\lambda x.diff(\lambda y.sin(F(x,y)),x))\ of\ true : true \to^? X_1$

$$
\begin{array}{l}
\quad \Rightarrow_{Case\ Eval} \\
case\ \lambda x.diff(\lambda y.sin(F(x,y)),x)\ of\ cos : true \rightarrow^{?} X_2, \\
\quad case\ X_2\ of\ true : true \rightarrow^{?} X_1 \\
\quad \Rightarrow_{Case\ Eval} \\
\lambda x.case\ \lambda y.sin(F(x,y))\ of \ldots, \lambda y.sin(G(x,y)) : \ldots, \ldots \rightarrow^{?} X_3, \\
\quad case\ X_3\ of\ cos : true \rightarrow^{?} X_2, case\ X_2\ of\ true : true \rightarrow^{?} X_1 \\
\quad \Rightarrow_{Case\ Select} \\
\lambda x.cos(F(x,x)) * diff(\lambda y.F(x,y),x) \rightarrow^{?} X_3, case\ X_3\ of\ cos : true \rightarrow^{?} X_2, \ldots \\
\quad \Rightarrow_{Bind} \\
case\ \lambda x.cos(F(x,x)) * diff(\lambda y.F(x,y),x)\ of\ cos : true \rightarrow^{?} X_2, \ldots \\
\quad \Rightarrow_{Case\ Eval} \\
\lambda x.case\ diff(\lambda y.F(x,y),x)\ of\ 1 : cos(F(x,x)), \ldots \rightarrow^{?} X_3', \ldots \\
\quad \Rightarrow_{Case\ Eval} \\
\lambda x.case\ \lambda y.F(x,y)\ of\ \lambda y.y : 1, \ldots \rightarrow^{?} X_4, \lambda x.case\ X_4(x)\ of\ 1 : cos(F(x,x)), \ldots \\
\quad \Rightarrow^{\{F \mapsto \lambda x,y.y\}}_{Projection} \\
\lambda x.case\ \lambda y.y\ of\ \lambda y.y : 1, \ldots \rightarrow^{?} X_4, \lambda x.case\ X_4(x)\ of\ 1 : cos(x), \ldots \rightarrow^{?} X_3', \ldots \\
\quad \Rightarrow_{Case\ Select} \\
\lambda x.1 \rightarrow^{?} X_4, \lambda x.case\ X_4(x)\ of\ 1 : cos(x), \ldots \rightarrow^{?} X_3', \ldots \\
\quad \Rightarrow_{Bind} \\
\lambda x.case\ 1\ of\ 1 : cos(x), \ldots \rightarrow^{?} X_3', case\ X_3'\ of\ cos : true \rightarrow^{?} X_2, \ldots \\
\quad \Rightarrow_{Case\ Select} \Rightarrow_{Bind} \Rightarrow_{Case\ Select} \Rightarrow_{Bind} \\
case\ true\ of\ true : true \rightarrow^{?} X_1 \quad \Rightarrow_{Case\ Select} \quad true \rightarrow^{?} X_1 \quad \Rightarrow_{Bind} \quad \{\}
\end{array}
$$

Thus, the computed solution is $\{F \mapsto \lambda x, y.y\}$.

6 Correctness and Completeness

As in the first-order case, we show completeness w.r.t. needed reductions. We first define needed reductions and then lift needed reductions to narrowing. In the following we assume an inductively sequential HRS $\mathcal{R}$ and assume LNT is invoked with the corresponding definitional trees.

For our purpose it is convenient to define needed reductions via LNT. Then we show that they are in fact needed. For modeling rewriting, the Guess rules are not needed: For LNT we have $S \stackrel{*}{\Rightarrow}{}^{\{\}}_{LNT} S'$ if and only if no Guess rules are used in the reduction. Hence no narrowing is performed. This can also be seen as an implementation of a particular rewriting strategy.

In order to relate a system of LNT goals to a term, we associate a position p with each case construct and a substitution θ for all newly introduced variables on the right. For each case expression $\mathcal{T} = case\ X\ of\ \ldots$ in a rule $\mathcal{T}' = f(\overline{X_n}) \rightarrow \mathcal{X}$ we attach the position p of X in the left-hand side of the corresponding rewrite rule. Formally, we define a function $l_{\mathcal{T}}$ such that $l_{\mathcal{T}}(f(\overline{X_n}) : \mathcal{X})$ yields the **labeled tree** for a rule $\mathcal{T} = f(\overline{X_n}) \rightarrow \mathcal{X}$:

- $l_{\mathcal{T}}(p_f : case\ X\ of\ \overline{\mathcal{T}_n}) = p_f : case_p\ X\ of\ \overline{l_{\mathcal{T}}(\mathcal{T}_n)}$
  where p is the position of X in $pat_{\mathcal{T}}(p_f : case\ X\ of\ \overline{\mathcal{T}_n})$
- $l_{\mathcal{T}}(p_f : r) = p_f : r$

We assume in the following that definitional trees for some inductively sequential HRS $\mathcal{R}$ are labeled.

The following invariant will allow us to relate a goal system with a term:

Theorem 4. *For an initial goal with* $case_\epsilon\ t\ of\ true : true \to^? X_1 \stackrel{*}{\Rightarrow}{}^{\{\}}_{LNT} S$, *$S$ is of one of the following two forms:*

1. $\lambda\overline{x}.case_{p_n}\ s\ of\ \ldots \to^? X_n, \lambda\overline{x}.case_{p_{n-1}}\ \lambda\overline{y}.X_n(\overline{x},\overline{y})\ of\ \ldots \to^? X_{n-1}, \ldots,$
   $\lambda\overline{x}.case_{p_2}\ \lambda\overline{y}.X_3(\overline{x},\overline{y})\ of\ \ldots \to^? X_2,\ case_{p_1}\ X_2\ of\ true : true \to^? X_1$
2. $r \to^? X_{n+1}, \lambda\overline{x}.case_{p_n}\ \lambda\overline{y}.X_{n+1}(\overline{x},\overline{y})\ of\ \ldots \to^? X_n,$
   $\lambda\overline{x}.case_{p_{n-1}}\ \lambda\overline{y}.X_n(\overline{x},\overline{y})\ of\ \ldots \to^? X_{n-1}, \ldots,$
   $\lambda\overline{x}.case_{p_2}\ \lambda\overline{y}.X_3(\overline{x},\overline{y})\ of\ \ldots \to^? X_2,\ case_{p_1}\ X_2\ of\ true : true \to^? X_1$

Furthermore, all $\overline{X_{n+1}}$ *are distinct and each variable* X_i *occurs only as shown above, i.e. at most twice in* $\ldots, e \to^? X_i, case\ X_i\ of \ldots$.

Notice that the second form in the above theorem is created by a Case Select rule application, which may reduce a case term to a non-case term, or by Case Eval with a rule $f(\overline{X_n}) \to r$. As only the Bind rule applies on such systems, they are immediately reduced to the first form. As we will see, the Bind rule corresponds to the replacement which is part of a rewrite step. Since we now know the precise form of goal systems which may occur, bound variables as arguments and binders are often omitted in goal systems for brevity.

The next goal is to relate LNT and rewriting. For a goal system S, we write $S{\downarrow}$ for the normal form obtained by applying Case Eval and Case Select.

Definition 5. We define an **associated substitution** for each goal system inductively on $\stackrel{*}{\Rightarrow}_{LNT}$:

- For an initial goal system of the form $S = case_\epsilon\ t\ of\ true : true \to^? X$, we define the associated substitution $\theta_S = \{X \mapsto t\}$.
- For the Case Eval rule on $S = \lambda\overline{x}.case_p\ \lambda\overline{y}.f(\overline{t})\ of \ldots \to^? X, G$ with

$$S \Rightarrow \lambda\overline{x},\overline{y}.\sigma(\mathcal{X}) \to^? X', \lambda\overline{x}.case_p\ \lambda\overline{y}.X'(\overline{x},\overline{y})\ of\ \ldots \to^? X, G\ =: S'$$

  we define $\theta_{S'} = \theta_S \cup \{X' \mapsto \lambda\overline{x}.(\theta_S X)|_p\}$.

For all other rules, the associated substitution is unchanged.

For a goal system S we write the associated substitution as θ_S. Notice that the associated substitution is not a "solution" as used in the completeness result and only serves to reconstruct the original term.

We can translate a goal system produced by LNT into one term as follows. The idea is that $case_p\ t\ of\ \ldots \to^? X$ should be interpreted as the replacement of the case term t at position p in $\theta_S X$, i.e., $(\theta_S X)[t]_p$. Extending this to goal systems yields the following definition:

Definition 6. For a goal system S of the form

$$[r \to^? X,]\ \lambda\overline{x}.case_{p_n}\ s\ of\ \ldots \to^? X_n, \ldots, case_{p_1}\ X_2\ of\ true : true \to^? X_1$$

(where $[r \to^? X,]$ is optional) with associated substitution θ we define the **associated term** $T(S)$ as $(\theta X_1)[(\theta X_2)[\ldots(\theta X_n(\overline{x}))[\theta s]_{p_n} \ldots]_{p_2}]_{p_1}$.

For instance, if we start with a goal system $S_1 = case_\epsilon\ t\ of\ true : true \rightarrow^? X$, then $T(S_1) = t$.

For a goal system S, we write $Bind(S)$ to denote the result of applying the Case Bind rule. Notice that the substitution of the Bind rule only affects the two leftmost goals.

Lemma 7. *Let $S = I(t)$. If $S{\downarrow}$ is of the form of Invariant 2, then $t = T(S{\downarrow})$ is reducible at position $p = p_1 \cdots p_n$. Furthermore, if $t \longrightarrow_p t'$, then $I(t'){\downarrow} = Bind(S{\downarrow}){\downarrow}$.*

Now, we can define needed reductions:

Definition 8. A term t has a needed redex p if $I(t){\downarrow}$ is of Invariant 2 with $p = p_1 \cdots p_n$.

It remains to show that needed reductions are indeed needed to compute a constructor headed term.

Theorem 9. *If t reduces to true, then t has a needed redex at position p and t must be reduced at p eventually. Otherwise, t is not reducible to true.*

The next desirable result is to show that needed reductions are normalizing. This is suggested from related works [15, 11], but is beyond the scope of this paper.

For a goal system S, we call the variables that do not occur in $T(S)$ **dummies**. In particular, all variables on the right and all variables in selectors in patterns of some tree in S are dummies.

Lemma 10. *If $S \stackrel{*}{\Rightarrow}{}^{\theta}_{LNT} \{\}$, then $\theta S \stackrel{*}{\Rightarrow}{}^{\{\}}_{LNT} \{\}$.*

Theorem 11 (Correctness of LNT). *If $I(t) \stackrel{*}{\Rightarrow}{}^{\theta}_{LNT} \{\}$ for a term t, then $\theta t \stackrel{*}{\longrightarrow} true$.*

We first state completeness in terms of LNT reductions.

Lemma 12. *If $\theta S \stackrel{*}{\Rightarrow}{}^{\{\}}_{LNT} \{\}$ and θ is in $\mathcal{R}$-normal form and contains no dummies of S,[6] then $S \stackrel{*}{\Rightarrow}{}^{\theta'}_{LNT} \{\}$ with $\theta' \leq \theta$.*

Theorem 13 (Completeness of LNT). *If $\theta t \stackrel{*}{\longrightarrow} true$ and θ is in $\mathcal{R}$-normal form, then $I(t) \stackrel{*}{\Rightarrow}{}^{\theta'}_{LNT} \{\}$ with $\theta' \leq \theta$.*

7 Optimality regarding Solutions

We show here another important aspect, namely uniqueness of the solutions computed. Compared to the more general case in [19], optimality of solutions is possible here, since we only evaluate to constructor-headed terms. For this to hold for all subgoals in a narrowing process, our requirement of constructor-based rules is also essential. For these reasons, we never have to chose between Case Select and Case Eval in our setting and optimality follows easily from the corresponding result of higher-order unification.

[6] I.e., $\mathcal{FV}(\theta) \cap \mathcal{FV}(S) = \mathcal{FV}(T(S))$

Theorem 14 (Optimality). *If $I(t) \overset{*}{\Rightarrow}{}^{\theta}_{LNT} \{\}$ and $I(t) \overset{*}{\Rightarrow}{}^{\theta'}_{LNT} \{\}$ are two different derivations, then θ and θ' are incomparable.*

It is also conjectured that our notion of needed reductions is optimal (this is subject to current research [16, 15, 3]). Note, however, that sharing is needed for optimality, as shown for the first-order case in [2].

8 Avoiding Function Synthesis

Although the synthesis of functional objects by full higher-order unification in LNT is very powerful, it can also be expensive and operationally complex. There is an interesting restriction on rewrite rules which entails that full higher-order unification is not needed in LNT for (quasi) first-order goals.

We show that the corresponding result in [4] is easy to see in our context, although lifting over binders obscures the results somewhat unnecessarily. Lifting may instantiate a first-order variable by a higher-order one, but this is only needed to handle the context correctly.

A term t is **quasi first-order** if t is a higher-order pattern without free higher-order variables. A rule $f(\overline{X_n}) \to \mathcal{X}$ is called **weakly higher-order**, if every higher-order free variable which occurs in $\mathcal{X}$ is in $\{\overline{X_n}\}$. In other words, higher-order variables may only occur directly below the root and are immediately eliminated when *hdt*s are introduced in the Case Eval rule.

Theorem 15. *If $I(t) \overset{*}{\Rightarrow}_{LNT} S$ where t is quasi first-order w.r.t. weakly higher-order rules, then $T(S)$ is quasi first-order.*

As a trivial consequence of the last result, Function Guess and Projection do not apply and Imitation is only used as in the first-order case.

9 Conclusions

We have presented an effective model for the integration of functional and logic programming with completeness and optimality results. Since we do not require terminating rewrite rules and permit higher-order logical variables and λ-abstractions, our strategy is a suitable basis for truly higher-order functional logic languages. Moreover, our strategy reduces to an optimal first-order strategy if the higher-order features are not used. Further work will focus on adapting the explicit model for sharing using goal systems from [19] to this refined context.

References

1. S. Antoy. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.
2. 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.

3. Andrea Asperti and Cosimo Laneve. Interaction systems I: The theory of optimal reductions. *Mathematical Structures in Computer Science*, 4:457–504, 1994.
4. 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.
5. Hendrik Pieter Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North Holland, 2nd edition, 1984.
6. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A logic plus functional language. *Journal of Computer and System Sciences*, 42(2):139–185, 1991.
7. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
8. M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation*, pages 252–266. Springer LNCS 1048, 1995.
9. M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. Technical report 96-2, RWTH Aachen, 1996.
10. J.R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.
11. Jan Willem Klop. *Combinatory Reduction Systems*. Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.
12. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1:497–536, 1991.
13. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.
14. Tobias Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349, 1991.
15. Vincent van Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, 1994. Amsterdam.
16. Vincent van Oostrom. Higher-order families, 1996. In this volume.
17. Christian Prehofer. Higher-order narrowing. In *Proc. Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 507–516. IEEE Computer Society Press, 1994.
18. 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.
19. Christian Prehofer. *Solving Higher-order Equations: From Logic to Programming*. PhD thesis, TU München, 1995. Also appeared as Technical Report I9508.
20. J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.