Higher-Order Narrowing with Convergent Systems

Christian Prehofer*

Technische Universität München**

Abstract. Higher-order narrowing is a general method for higher-order equational reasoning and serves for instance as the foundation for the integration of functional and logic programming. We present several refinements of higher-order lazy narrowing for convergent (terminating and confluent) term rewrite systems and their application to program transformation. The improvements of narrowing include a restriction of narrowing at variables, generalizing the first-order case. Furthermore, functional evaluation via normalization is shown to be complete and a partial answer to the eager variable elimination problem is presented.

1 Introduction and Overview

Higher-order narrowing is a method for solving higher-order equations modulo a set of rewrite rules. It forms the basis of functional-logic programming and has been extensively studied in the first-order case, for a survey see [10]. Motivated by functional programming, there exist several higher-order extensions for such languages [7, 18, 32]. Even more expressive than the latter is the language Escher, proposed in [17]. Higher-order narrowing [29] can be used as an operational semantics for such languages. The basis for narrowing are higher-order rewrite rules. Examples are the function *map* with

$$map(F, [X|R]) \rightarrow [F(X)|map(F, R)]$$

or a rule for pushing quantifiers inside:

$$\forall x.P \wedge Q(x) \rightarrow P \wedge \forall x.Q(x)$$

In the latter example the quantifier $\forall$ is a constant of type $(term \rightarrow bool) \rightarrow bool$, where $\forall(\lambda x.P)$ is written as $\forall x.P$ for brevity. For more examples on higher-order rewriting, we refer to [30], to [24] for formalizing logics and λ-calculi, and for Process Algebras to [27].

With higher-order narrowing we solve higher-order unification problems modulo such rewrite rules. Compared to the first-order case, also values for functional

* Research supported by the DFG under grant Br 887/4-2, *Deduktive Programmentwicklung* and by ESPRIT WG 6028, *CCL*.

** Full Address: Fakultät für Informatik, Technische Universität München, 80290 München, Germany. E-mail: prehofer@informatik.tu-muenchen.de

variables have to be computed via higher-order unification. To show the expressiveness of this method, we give an example for program transformation.

The framework for higher-order narrowing in [29] serves as a basis for the refinements of lazy narrowing we present here. For convergent higher-order rewrite systems, we show several techniques that use the determinism of convergent systems. The main contributions are as follows:

- We disallow narrowing at variable positions, generalizing the first-order case, as it is possible to restrict attention to R-normalized solutions. This is the gist of (first-order) narrowing, since narrowing into variables is undesirable.
- Simplification of equational goals via rewriting is shown to be complete. This is an important refinement as it performs deterministic evaluation without any search.
- Completeness of eager variable elimination (see below) is an open problem even for the first-order case [33]. By using oriented goals, this can be partially solved.
- Several deterministic operations for constructors, i.e. uninterpreted symbols, are shown.

Notice that the third item is also new for the first-order case. Another partial solution to this problem has been presented recently [20]. The significance of the other contributions has been argued in the first-order case in several papers, for references see [10].

Eager variable elimination means to solve a goal $X =^? t$ by binding X to t, without considering alternative rules. The result for eager variable elimination is based on a simpler notion of goals to be solved: we consider *oriented* equational goals of the form $s \rightarrow^? t$, where a substitution θ is a solution if $\theta s \xrightarrow{*} \theta t$. We show that for goals of the form $X \rightarrow^? t$, elimination is complete. We adopt this simpler operational model, which also eases technicalities, with no loss of expressiveness.

The higher-order case is more subtle in many respects. One of the typical technical problems is that higher-order substitutions and reducibility wrt a rewrite system R are harder to relate. For instance, if θt is R-normalized, then neither θ nor t must be R-normalized, which is the basis for first-order narrowing. The solution is to use patterns, a restricted class of λ-terms, for the left-hand sides of rules. This is no limitation in practice and allows to argue similar to the first-order case when needed.

The paper is organized as follows. Section 3 introduces a calculus for higher-order narrowing that utilizeses normalized solutions. This is followed by an analysis of deterministic operations for constructors in Section 4 and deterministic variable elimination in Section 5. Narrowing with simplification is the subject of Section 6. An application to program transformation is shown in Section 7.

2 Preliminaries

We briefly introduce simply typed λ-calculus (see e.g. [12]). 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 and are denoted by $\{\overline{X_n \mapsto t_n}\}$. Free and bound variables of a term t will be denoted as $\mathcal{FV}(t)$ and $\mathcal{BV}(t)$, respectively. 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 $\lambda x.(tx) =_\eta t$.

For β-conversion (η-conversion), applying the rule from left to right is called β-reduction (η-reduction), and expansion in the other direction. A term is in $\beta\eta$-normal form if no β- or η-reductions apply, and η-expanded if no η-expansion applies. The **long $\beta\eta$-normal form** of a term t, denoted by $\updownarrow^\eta_\beta t$, 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 $\updownarrow^\eta_\beta s =_\alpha \updownarrow^\eta_\beta t$. 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.

The convention that α-equivalent terms are identified and that free and bound variables are kept disjoint (see also [2]) 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 [21]. Also, the

unification of a linear pattern with a second-order term is decidable and finitary, if they are variable-disjoint [28].

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 $\lambda x.G(H(x))$.

2.1 Higher-Order Rewriting

The following definitions for higher-order rewriting are in the lines of [24, 19].

Definition 1. A **rewrite rule** is a pair $l \to r$ such that l is a pattern but not η-equivalent to a free variable, l and r are long $\beta\eta$-normal forms of the same base type, and $\mathcal{FV}(l) \supseteq \mathcal{FV}(r)$. A **Higher-Order Rewrite System** (**HRS**) is a set of rewrite rules. The letter R always denotes an HRS. Assuming a rule $(l \to r) \in R$ 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 instance, with the quantifier rule of the first section, we have the following rewrite step:

$$\forall y.\forall x.p(y) \wedge q(x, y) \longrightarrow^{\forall x.P \wedge Q(x) \to P \wedge \forall x.Q(x)} \forall y.p(y) \wedge \forall x.q(x, y)$$

For a rewrite step we often omit some of the parameters $l \to r, p$ and θ. We assume that constants 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** if no rule from R applies and a substitution θ is R-**normalized** if if all terms in the image of θ are in R-normal form.

In contrast to the first-order notion of term rewriting, $\longrightarrow$ is not stable under substitution: reducibility of s does not imply reducibility of θs. Its transitive reflexive closure is however stable [19]:

Lemma 2. *Assume an GHRS R. If $s \stackrel{*}{\longrightarrow}^R t$, then $\theta s \stackrel{*}{\longrightarrow}^R \theta t$.*

A reduction is called **confluent**, if any two reductions from a term t are joinable, i.e. if $t \stackrel{*}{\longrightarrow} u$ and $t \stackrel{*}{\longrightarrow} v$ then there exists w with $u \stackrel{*}{\longrightarrow} w$ and $v \stackrel{*}{\longrightarrow} w$. For results on confluence of higher-order rewrite systems, we refer to [19]. A terminating and confluent reduction system is called **convergent**.

Termination orderings for higher-order rewriting can be found in [27, 16]. For our purpose, we need the following result, which can be shown similar to the first-order case [15]. A term $s = \lambda\overline{x_n}.s_0$ is a **subterm modulo binders** of $t = \lambda\overline{x_n}.t_0$, written as $s <_{sub} t$, if s_0 is a true subterm of t_0

Theorem 3. *The reduction $\longrightarrow^R_{sub} = \longrightarrow^R \cup >_{sub}$ is terminating for a GHRS R if $\longrightarrow^R$ is terminating.*[3]

[3] All missing proofs can be found in [30].

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.

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'.

2.2 Higher-Order Unification

We introduce in the following the transformations for higher-order unification as in [34]. Although higher-order unification is undecidable in general, it performs remarkably well in systems such as λ-Prolog [22] and Isabelle [25]. For programming applications, there even exist decidable fragments [28, 30].

In contrast to first-order unification, we solve unification problems modulo the conversions of λ-calculus, i.e. θ is a unifier of $s =^? t$ if $\theta s =_{\alpha\beta\eta} \theta t$. We examine in the following the most involved case of higher-order unification: flex-rigid goals of the form $\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.v(\overline{t'_m})$, where v is not a free variable. Clearly, for any solution θ to F the term $\theta F(\overline{t_n})$ must have (after β-reduction) the symbol v as its head. There are two possibilities:

- In the first case, v already occurs in (the solution to) some t_i. For instance, consider the equation $F(a) =^? a$, where $\{F \mapsto \lambda x.x\}$ is a solution based on a **projection**. In general, a projection binding for F is of the from $\{F \mapsto \lambda\overline{x_n}.x_i(\ldots)\}$. As some argument, here a, is carried to the head of the term, such a binding is called projection.
- In the second case, the head of the solution to F is just the desired symbol v. For instance, in the last example, an alternative solution is $\{F \mapsto \lambda x.a\}$. This is called **imitation.** Notice that imitation is not possible if v is a bound variable.

To solve a flex-rigid pair, the strategy is to guess an appropriate imitation or projection binding only for one rigid symbol, here a, and thus approximate the solution to F. Unification proceeds by iterating this process which focuses only on the outermost symbol. Roughly speaking, the rest of the solution for F is left open by introducing new variables.

Definition 4. Assume an equation $\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.v(\overline{t'_m})$, where all terms are in long $\beta\eta$-normal form. An **imitation binding** for F is of the form

$$F \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})$$

where $\overline{H_m}$ are new variables of appropriate type. A **projection binding** for F is of the form

$$F \mapsto \lambda\overline{x_n}.x_i(\overline{H_p(\overline{x_n})})$$

where $\overline{H_p}$ are new variables with $\overline{H_p : \tau_p}$ and $x_i : \overline{\tau_p} \to \tau$. A **partial binding** is an imitation or a projection binding.

Notice that in the above definition, the bindings are not written in long $\beta\eta$-normal form. The long $\beta\eta$-normal form of an imitation or projection binding can be written as

$$F \mapsto \lambda\overline{x_n}.v(\overline{\lambda\overline{z_{j_p}}.H_p(\overline{x_n},\overline{z_{j_p}})}).$$

A full exhibition of the the types involved can be found in [34].

For lack of space, the transformation rules for higher-order unification are shown in Figure 1 together with the narrowing rules. The rules consist of the basic rules for unification, such as Deletion, Elimination and Decomposition plus the two rules explained above: Imitation and Projection. For the purpose of narrowing (to be detailed later), the rules work on oriented goals, which does not affect unification, and use subscripts (d), which only serve to improve narrowing.

It should be mentioned that the higher-order unification rules only perform so-called pre-unification. The idea of pre-unification is to handle **flex-flex pairs** as constraints and not to attempt to solve them explicitly. These are equations of the form $\lambda\overline{x_k}.P(\ldots) =^? \lambda\overline{x_k}.P'(\ldots)$. Huet [13] showed that for such pairs there may exist an infinite chain of unifiers, one more general than the other, without any most general one. Since flex-flex pairs are guaranteed to have at least one unifier, e.g. $\{P \mapsto \lambda\overline{x_m}.a, P' \mapsto \lambda\overline{x_n}.a\}$, pre-unification is sufficient.

3 Lazy Narrowing with Normalized Solutions

We introduce in this section higher-order lazy narrowing and refine it for R-normalized solutions. Consider a solution θ of an equational goal $s \to^? t$ with $\theta s \xrightarrow{*} \theta t$.[4] For any solution there exists an equivalent R-normalized one, assuming convergent rewrite systems. Hence it is a desirable restriction to consider only these. In the higher-order case, narrowing at (sub-)terms with variable heads such as $H(t)$ is needed [29]. The main improvement we discuss in this section is that narrowing is not needed at goals of the form $H(\overline{x_n}) \to^? t$ for normalized solutions, which covers many practical cases. The rules of System LNN for lazy higher-order narrowing, shown in Figure 1, consist of the rules for higher-order unification plus two narrowing rules; they are a refinement of System LN in [29].

Let $s \overset{?}{\leftrightarrow} t$ stand for one of $s \to^? t$ and $t \to^? s$. For a sequence $\Rightarrow^{\theta_1} \ldots \Rightarrow^{\theta_n}$ of LNN steps, we write $\overset{*}{\Rightarrow}^{\theta}$, where $\theta = \theta_n \ldots \theta_1$.

The subscripts (d) and d on goals only serve for a particular optimization and are not needed for soundness or completeness. The idea is to use **marked goals** $s \to^?_d t$. These are created only in the last rule, in order to avoid repeated application of Lazy Narrowing rules on these goals. The remaining rules work on both marked goals and unmarked goals, indicated by $\to^?_{(d)}$. For both $\overset{?}{\leftrightarrow}$ and $\to^?_{(d)}$ the rules are intended to preserve the orientation for $\overset{?}{\leftrightarrow}$ and marking for $\to^?_{(d)}$. Only the Decomposition rule and the Imitation rule, which includes decomposition,

[4] Although this corresponds only to equational matching, an equational unification problem $s =_R t$ can easily be encoded by adding a new rule $X =_R X \to \mathit{true}$ and solving the goal $s =_R t \to^? \mathit{true}$.

Deletion

$$\{t \to^{?}_{(d)} t\} \cup S \;\Rightarrow\; S$$

Decomposition

$$\{\lambda\overline{x_k}.f(\overline{t_n}) \to^{?}_{(d)} \lambda\overline{x_k}.f(\overline{t'_n})\} \cup S \;\Rightarrow\; \{\overline{\lambda\overline{x_k}.t_n \to^{?} \lambda\overline{x_k}.t'_n}\} \cup S$$

Elimination

$$\{F \overset{?}{\leftrightarrow}_{(d)} \lambda\overline{x_k}.t\} \cup S \Rightarrow^{\theta} \theta S \text{ if } F \notin \mathcal{FV}(\lambda\overline{x_k}.t) \text{ and where } \theta = \{F \mapsto \lambda\overline{x_k}.t\}$$

Imitation

$$\{\lambda\overline{x_k}.F(\overline{t_n}) \overset{?}{\leftrightarrow}_{(d)} \lambda\overline{x_k}.f(\overline{t'_m})\} \cup S \Rightarrow^{\theta} \overline{\{\lambda\overline{x_k}.H_m(\overline{\theta t_n}) \overset{?}{\leftrightarrow} \lambda\overline{x_k}.\theta t'_m\}} \cup \theta S$$
where $\theta = \{F \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\}$
is an imitation binding with fresh variables

Projection

$$\{\lambda\overline{x_k}.F(\overline{t_n}) \overset{?}{\leftrightarrow}_{(d)} \lambda\overline{x_k}.v(\overline{t'_m})\} \cup S \Rightarrow^{\theta} \{\lambda\overline{x_k}.\theta t_i(\overline{H_j(\overline{t_n})}) \overset{?}{\leftrightarrow}_{(d)} \lambda\overline{x_k}.v(\overline{\theta t'_m})\} \cup \theta S$$
where $\theta = \{F \mapsto \lambda\overline{x_n}.x_i(\overline{H_j(\overline{x_n})})\}$,
is a projection binding with fresh variables

Lazy Narrowing with Decomposition

$$\{\lambda\overline{x_k}.f(\overline{t_n}) \to^{?} \lambda\overline{x_k}.t\} \cup S \;\Rightarrow\; \{\overline{\lambda\overline{x_k}.t_n \to^{?} \lambda\overline{x_k}.l_n}\} \cup \{\lambda\overline{x_k}.r \to^{?} \lambda\overline{x_k}.t\} \cup S$$
where $f(\overline{l_n}) \to r$ is an $\overline{x_k}$-lifted rule

Lazy Narrowing at Variable

$$\{\lambda\overline{x_k}.H(\overline{t_n}) \to^{?} \lambda\overline{x_k}.t\} \cup S \;\Rightarrow\; \{\lambda\overline{x_k}.H(\overline{t_n}) \to^{?}_{d} \lambda\overline{x_k}.l\} \cup \{\lambda\overline{x_k}.r \to^{?} \lambda\overline{x_k}.t\} \cup S$$
where $\overline{x_k}.H(\overline{t_n})$ is not a pattern and
$l \to r$ is an $\overline{x_k}$-lifted rule

Fig. 1. System LNN for Lazy Narrowing

transform marked goals to unmarked goals. In other words, on marked goals the Lazy Narrowing rules may only be applied after some decomposition took place.

Consider for instance the matching problem $\lambda x.H(f(x)) \to^{?} \lambda x.h(g(x), f(x))$, modulo the rule $f(f(X)) \to g(X)$. Here LNN yields

$$\{\lambda x.H_1(f(x)) \to^{?} \lambda x.g(x), \lambda x.H_2(f(x)) \to^{?} \lambda x.f(x)\}$$

by the imitation $\{H \mapsto \lambda y.h(H_1(y), H_2(y))\}$. Then the second goal can be solved by Projection, and the first by Lazy Narrowing to

$$\{\lambda x.H_1(f(x)) \to_d^? \lambda x.f(f(X(x))), \lambda x.g(X(x)) \to^? \lambda x.g(x)\}$$

Notice that the rewrite rule has been lifted over the variable x in the binding environment. As the first goal is marked, Lazy Narrowing does not re-apply. This is an important restriction and improves the similar system in [29], as otherwise infinite reductions occur, as in this case, very often. The two goals can be solved by several higher-order unification steps, which yield the solution

$$\{H_1 \mapsto \lambda y.f(y), X \mapsto \lambda x.x\}.$$

Observe that the last two rules of LNN can be integrated into one rule of the form

$$\{\lambda\overline{x_k}.s \to^? \lambda\overline{x_k}.t\} \cup S \Rightarrow \{\lambda\overline{x_k}.s \to_d^? \lambda\overline{x_k}.f(\overline{l_n}), \lambda\overline{x_k}.r \to^? \lambda\overline{x_k}.t\} \cup S,$$

which is used in the completeness proofs and is called the **Lazy Narrowing** rule. From this rule, the narrowing rules of LNN can easily be derived, e.g. the first by decomposition on f.

Theorem 5. *If $s \to^? t$ has solution θ, i.e. $\theta s \xrightarrow{*}{}^R \theta t$, and θ is R-normalized for a convergent HRS R, then $\{s \to^? t\} \Rightarrow_{LNN}^{\delta} F$ such that δ is more general, modulo the newly added variables, than θ and F is a set of flex-flex goals.*

The proof proceeds as in the more general Theorem 9, which we show later. A key ingredient is the following lemma which generalizes the first-order case:

Lemma 6. *Assume an HRS R and a substitution θ. Then $\theta F(\overline{x_n})$ is R-reducible, iff θF is R-reducible.*

4 Deterministic Narrowing Rules for Constructors

In practice, rewrite systems often have a number of symbols, called constructors, that only serve as data structures. For constructor symbols, we give a few simple additional rules for Lazy Narrowing in Figure 2. Their main advantage is that their application is deterministic. The rules cover the cases where the root symbol of the left side of a goal is a constructor. Notice that the rules, except for the first, are only possible with oriented goals, where evaluation proceeds only from left to right. The correctness of the rules follows easily.

Deterministic Constructor Decomposition

$$\{\lambda\overline{x_k}.c(\overline{t_n}) \rightarrow^?_{(d)} \lambda\overline{x_k}.c(\overline{t'_n})\} \cup S \;\Rightarrow\; \{\overline{\lambda\overline{x_k}.t_n \rightarrow^? \lambda\overline{x_k}.t'_n}\} \cup S$$
if c is a constructor symbol

Deterministic Constructor Imitation

$$\{\lambda\overline{x_k}.c(\overline{t_n}) \rightarrow^?_{(d)} \lambda\overline{x_k}.F(\overline{x_m})\} \cup S \Rightarrow^{\theta} \{\overline{\lambda\overline{x_k}.t_n \rightarrow^? \lambda\overline{x_k}.H_n(\overline{x_m})}\} \cup \theta S$$
where $\theta = \{F \mapsto \lambda\overline{x_m}.f(\overline{H_n(\overline{x_m})})\}$
is an imitation binding with fresh variables

Constructor Clash

$$\{\lambda\overline{x_k}.c(\overline{t_n}) \rightarrow^?_{(d)} \lambda\overline{x_k}.v(\overline{t'_n})\} \cup S \;\Rightarrow\; fail$$
if $c \neq v$, where c is a constructor symbol
and v is not a free variable

Fig. 2. Deterministic Constructor Rules

5 Deterministic Variable Elimination

Eager variable elimination is a particular strategy of general E-unification systems. The idea is to apply the elimination rule as a deterministic operation whenever possible. It is an open problem of general (first-order) E-unification strategies if eager variable elimination is complete [33].

In our case, with oriented goals, we obtain more precise results by differentiating the orientation of the goal to be eliminated. We distinguish two cases of variable elimination, where in one case elimination is deterministic, i.e. no other rules have to be considered for completeness.

Theorem 7. *System LNN with eager variable elimination on goals of the form $X \rightarrow^? t$ with $X \notin \mathcal{FV}(t)$ is complete for a convergent HRS R.*

The main idea of the proof is that there can be no rewrite step in $\theta X \rightarrow^? \theta t$, thus we have $\theta X = \theta t$, assuming that θ is R-normalized.

In the general case, variable elimination may copy reducible terms with the result that the reductions have to be performed several times. Notice that this case of variable elimination does not affect the reductions in the solution considered, as only terms in normal form are copied: θt must be in normal form.

There are however a few important cases when elimination on goals of the form $t \rightarrow^? X$ is deterministic [30]: if t is either ground and in R-normal form or a pattern without defined symbols. Furthermore, for left-linear rewrite systems, elimination on goals of the form $t \rightarrow^? X$ is not needed, as shown in [30].

6 Lazy Narrowing with Simplification

Simplification by normalization of goals is one of the earliest [4] and one of the most important optimizations. Its motivation is to prefer deterministic reduction over search within narrowing. Notice that normalization coincides with deterministic evaluation in functional languages. For first-order systems, functional-logic programming with normalization has shown to be a more efficient control regime than pure logic programming [6, 9].

The main problem of normalization is that completeness of narrowing may be lost. For first-order (plain) narrowing, there exist several works dealing with completeness of normalization in combination with other strategies (for an overview see [10]). Recall from Section 4 that 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 $f(1) \to 1$, we can simplify a goal $f(1) \to^? g(Y)$ by $\{f(1) \to^? g(Y), \ldots\} \Rightarrow \{1 \to^? g(Y), \ldots\}$ and deterministically detect a failure.

In the following, we show completeness of simplification for lazy narrowing. The result is similar to the corresponding result for the first-order case [11]. The technical treatment here is more involved in many respects due to the higher-order case. Using oriented goals, however, simplifies the completeness proof.

For oriented goals, normalization is only complete for goals $s \to^? t$, 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 8. Normalizing Lazy Narrowing, called NLN, is defined as the rules of LNN plus arbitrary simplification steps on goals. A **simplification step** on a goal $s \to^? t$ is a rewrite step on s, written as $\{s \to^? t\} \Rightarrow_{NLN} \{s' \to^? t\}$ if $s \longrightarrow^R s'$.

We first need an auxiliary construct for the termination ordering in the completeness result. The **decomposition function** D is defined as

$$D(s \to^? t) = s \to^? t$$
$$D(\lambda\overline{x_k}.f(\overline{s_n}) \to^?_d \lambda\overline{x_k}.f(\overline{t_n})) = \overline{\lambda\overline{x_k}.s_n \to^? \lambda\overline{x_k}.t_n}$$

and is undefined otherwise. The function D extends component-wise to sets of goals. The idea of D is to view marked goals as goals with delayed decomposition.

Theorem 9 Completeness of NLN. *Assume a confluent HRS R that terminates with order $<^R$. If $s \to^? t$ has solution θ, i.e. $\theta s \stackrel{*}{\longrightarrow}^R \theta t$ where θt and θ are R-normalized, then $\{s \to^? t\} \stackrel{*}{\Rightarrow}^{\delta}_{NLN} F$ such that δ is more general modulo the newly added variables than θ and F is a set of flex-flex goals.*

Proof. Let $<^R_{sub} = <^R \cup <_{sub}$. Assume $\overline{G_n} = \overline{s_n \to^?_{(d)} t_n}$ is a system of goals with solution θ, i.e. $\overline{\theta s_n \stackrel{*}{\longrightarrow}^R \theta t_n}$. Let $\overline{s'_m \to^? t'_m} = D(\overline{G_n})$. The proof proceeds by induction on the following lexicographic termination order on $(\overline{G_n}, \theta)$:

- A: $<^R_{sub}$ extended to the multiset of $\{\overline{\theta s'_m}\}$,
- B: multiset of sizes of the bindings in θ,
- C: multiset of sizes of the goals $\overline{\theta G_n}$,
- D: $<^R$ extended to the multiset of $\{\overline{s_n}\}$.

By Theorem 3, item A is terminating. For the proof we need two invariants: first, all $\overline{t'_m}$ are R-normalized terms. Secondly, for marked goals $s \rightarrow^?_d t$, $Head(\theta s) = Head(\theta t)$ is not a free variable and furthermore, no rewrite step at root position occurs in $\theta s \stackrel{*}{\longrightarrow}^R \theta t$. Except for the narrowing rule, it follows easily that the latter is invariant. E.g. Decomposition and Imitation on marked goals decompose the outermost symbol and yield unmarked goals.

In the following we show that normalization reduces this ordering and, furthermore, that for a non flex-flex goal some rule applies that reduces the ordering. In addition, we show in each of these cases that the above invariants are preserved. First, we select some non flex-flex goal $s \rightarrow^? t$ from $\overline{G_n}$; if none exists, the case is trivial.

We first consider the case where a simplification step is applied to an unmarked goal, i.e. $s \rightarrow^? t$ is transformed to $s' \rightarrow t$. We obtain $\theta s \stackrel{*}{\longrightarrow} \theta s'$ from Lemma 2. As θt is in normal form, confluence of R yields $\theta s \stackrel{*}{\longrightarrow} \theta s' \stackrel{*}{\longrightarrow} \theta t$. Thus θ is a solution of $s' \rightarrow t$. For termination, we have two cases:

- If $\theta s = \theta s'$, measures A through C remain unchanged, whereas D decreases.
- If $\theta s \neq \theta s'$ measure A decreases.

Clearly, the invariants are preserved.

If no simplification is applied, we distinguish two cases: if $\theta s = \theta t$, then we proceed as in pure higher-order unification, as one of the rules of higher-order unification must apply. In case of the Deletion rule, measure A decreases. For Decomposition on marked goals, A and B remain unchanged, whereas C decreases. On unmarked goals, Decomposition reduces A. Imitation on marked goals does not change A, but reduces B; on unmarked goals, it reduces A. Projection only decreases B.

Normalization of the associated solution is preserved in these cases: In case of a Projection or Imitation, the partial binding computed maps a variable X to a higher-order pattern of the form $\lambda\overline{x_n}.v(\overline{H_m(\overline{x_n})})$. The new, intermediate solution constructed maps the newly introduced variables $\overline{H_m(\overline{x_n})}$ to subterms of θX, which are in R-normal form. Hence all $\overline{\theta H_m}$ must be in R-normal form. For the elimination rule, no new variables are introduced, thus the solution remains R-normalized.

Furthermore, the terms $\overline{\theta t'_m}$ do not change under Decomposition and Imitation on marked goals. On unmarked goals, Decomposition and Imitation yield new right hand sides t_n. These are subterms of θt and are thus R-normalized.

In the remaining case, there must be a rewrite step in $\theta s \stackrel{*}{\longrightarrow} \theta t$. First, assume there is no rewrite step at the root position in $\theta s \stackrel{*}{\longrightarrow} \theta t$. Hence all terms in this sequence have the same root symbol. Then similar to the last case, one of the unification rules must apply.

Now consider the case with rewrite steps in $\theta s \xrightarrow{*} \theta t$ at root position. Clearly, $s \rightarrow^? t$ cannot be marked. Further, s cannot be of the form $\lambda\overline{x_n}.X(\overline{y_m})$: with the invariant that θ' is R-normalized, it is clear that there can be no rewrite step in the solution θ of a goal $\lambda\overline{x_n}.X(\overline{y_m}) \rightarrow^? t$ as θX is in R-normal form.

Assume the first rewrite step in $\theta s \xrightarrow{*} \theta t$ is $\theta s \xrightarrow{*} \lambda\overline{y_k}.s_1 \longrightarrow_\epsilon^{l\rightarrow r} \lambda\overline{y_k}.t_1$, with the rule $l \rightarrow r$. Notice that $s_1 \rightarrow t_1$ must be an instance of $l \rightarrow r$ (modulo lifting).

We apply Lazy Narrowing (integrating the two lazy narrowing rules), yielding the subgoals:

$$s \rightarrow_d^? \lambda\overline{y_k}.l, \lambda\overline{y_k}.r \rightarrow^? t$$

As there exists δ such that $s_1 = \delta l$ and $t_1 = \delta r$, we can extend θ to the newly added variables: define $\theta' = \theta \cup \delta$. Let $\overline{s_m \rightarrow^? l_m} = D(\theta' s \rightarrow_d^? \lambda\overline{y_k}.\theta' l)$. Clearly, $s_i <_{sub}^R \theta' s$ holds, and $\theta'\lambda\overline{x_k}.r <_{sub}^R \theta' s$ follows from $\theta' s \xrightarrow{*} \theta' r$. Thus θ' is a solution of $\overline{s_m \rightarrow^? l_m}$ and $r \rightarrow^? t$ that coincides with θ on $\mathcal{FV}(\overline{G_n})$. It remains to show that θ' is in R-normal form. As the reduction is innermost, all $\theta\overline{l_m}$ are in R-normal form. As l is a pattern, this yields that θ' is R-normalized. Since we consider the first rewrite step a root position, the new marked goal $s \rightarrow_d^? \lambda\overline{x_k}.l$ fulfills the invariant, as $Head(\theta s) = Head(\theta l)$ and no rewrite step can occur at root position. □

The termination ordering in this proof is rather complex. For instance, the last item in the ordering is needed for the following example: assume a goal $\lambda x.c(F(x,t)) \rightarrow^? \lambda x.c(x)$ with solution $\theta = \{F \mapsto \lambda x, y.x\}$. Here, normalization of t does not change $\theta\lambda x.c(F(x,t))$.

7 An Example: Program Transformation

The utility of higher-order unification for program transformations has been shown nicely by Huet and Lang [14] and has been developed further in [26, 8]. The following models an example for unfold/fold program transformation in [5]. We assume the following rules for lists:

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

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

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

that first maps F onto a list and then adds the elements. This simple implementation for g is very 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 desired behavior in a syntactic fashion by one simple equation:

$$\lambda f, x, l.g(f,[x|l]) =^? B(f(x), g(f,l))$$

The variable B represents the body of the function to be computed. The schema on the right only allows recursing on l for g, indicated by the argument $g(f,l)$ to B, and similarly allows to use $f(x)$. Notice that the bound variables above can be viewed as $\forall$-quantified variables.

To solve this equation, we add a rule $X =^? X \to true$, where we view $=^?$ as a new (infix) constant and then apply narrowing, yielding the solution $\theta = \{B \mapsto \lambda f x, rec.plus(fx, rec)\}$ 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

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

where narrowing with the newly added rule $X =^? X \to true$ yields the two goals

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

These can be solved by pure higher-order unification. Observe that simplification in this examples corresponds to (partial) evaluation.

8 Conclusions and Related Work

We have presented several refinements for narrowing, based on the determinism of reduction in convergent systems, in a highly expressive setting. The results apply to higher-order functional-logic programming, for which there exist several approaches and implementations [3, 7, 18, 32, 17] and to high-level reasoning, e.g. dealing with programs or mathematics [29]. Further development of higher-order narrowing towards functional-logic programming languages can be found in [30].

The work in [31] on higher-order narrowing considers only a restricted class of λ-terms, higher-order patterns with first-order equations, which does not suffice for modeling higher-order functional programs. The approach to higher-order narrowing in [1] aims at narrowing with higher-order functional programs, but restricts higher-order variables in the left-hand sides of rules and only permits first-order goals. These restrictions seem to be similar to the ones in [7].

Compared to higher-order logic programming [23], predicates and terms are not separated here. In the former, higher-order λ-terms are used for data structures and do not permit higher-order programming as in functional languages. For instance, the function *map* as in the last section cannot be written directly in higher-order logic programming.

References

1. 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*, Lecture Notes in Computer Science, vol. 845, München, Germany, 7–9 September 1994. Springer-Verlag.
2. Hendrik Pieter Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North Holland, 2nd edition, 1984.
3. P. G. Bosco and E. Giovannetti. IDEAL: An ideal deductive applicative language. In *Symposium on Logic Programming*, pages 89–95. IEEE Computer Society, The Computer Society Press, September 1986.
4. M. Fay. First order unification in equational theories. In *Proc. 4th Conf. on Automated Deduction*, pages 161–167. Academic Press, 1979.
5. Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, Wokingham, 1988.
6. L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Symposium on Logic Programming*, pages 172–184. IEEE Computer Society, Technical Committee on Computer Languages, The Computer Society Press, July 1985.
7. 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, *Computer Science Logic. Selected papers from CSL'92*, LNCS, pages 216–231, San Miniato, Italy, September 1992. Springer-Verlag.
8. 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.
9. 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.
10. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
11. M. Hanus. Lazy unification with simplification. In *Proc. 5th European Symposium on Programming*, pages 272–286. Springer LNCS 788, 1994.
12. J.R. Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.
13. Gérard Huet. *Résolution d'équations dans les languages d'ordre 1,2,...ω.* PhD thesis, University Paris-7, 1976.
14. Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
15. Jean-Pierre Jouannaud and Claude Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986.
16. Stefan Kahrs. Towards a domain theory for terminatin proofs. In *International Conference on Rewriting Techniques and Applications, RTA*, 1995. To appear.
17. John Wylie Lloyd. Combining functional and logic programming languages. In *Proceedings of the 1994 International Logic Programming Symposium, ILPS'94*, 1994.
18. Hendrik C.R Lock. *The Implementation of Functional Logic Languages.* Oldenbourg Verlag, 1993.

19. Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. Technical report, Institut für Informatik, TU München, 1994.
20. Aart Middeldorp, Satoshi Okui, and Tetsuo Ida. Lazy narrowing: Strong completeness and eager variable elimination. In *Proceedings of the 20th Colloquium on Trees in Algebra and Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.
21. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1:497–536, 1991.
22. Gopalan Nadathur and Dale Miller. An overview of λ-Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proc. 5th Int. Logic Programming Conference*, pages 810–827. MIT Press, 1988.
23. Gopalan Nadathur and Dale Miller. Higher-order logic programming. Technical Report CS-1994-38, Department of Computer Science, Duke University, December 1994. To appear in *Volume 5* of *Handbook of Logic in Artificial Intelligence and Logic Programming*, D. Gabbay, C. Hogger and A. Robinson (eds.), Oxford University Press.
24. Tobias Nipkow. Higher-order critical pairs. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 342–349, Amsterdam, The Netherlands, 15–18 July 1991. IEEE Computer Society Press.
25. Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.
26. 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.
27. Jaco van de Pol. Termination proofs for higher-order rewrite systems. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Higher-Order Algebra, Logic and Term Rewriting*, volume 816 of *Lect. Notes in Comp. Sci.*, pages 305–325. Springer-Verlag, 1994.
28. Christian Prehofer. Decidable higher-order unification problems. In *Automated Deduction — CADE-12*, LNAI 814. Springer-Verlag, 1994.
29. Christian Prehofer. Higher-order narrowing. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 507–516. IEEE Computer Society Press, 1994.
30. Christian Prehofer. *Solving Higher-order Equations: From Logic to Programming.* PhD thesis, TU München, 1995.
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, *Proceedings of the Seventh International Conference on Logic Programming*, pages 529–545, Jerusalem, 1990. The MIT Press.
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.

This article was processed using the LaTeX macro package with LLNCS style