J. Functional Programming **1** (1): 1–000, January 1999. Printed in the United Kingdom
© 1999 Cambridge University Press

Higher-Order Narrowing with Definitional Trees[1]

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

Christian Prehofer [2]
IC Networks, Siemens AG, Hofmannstr. 51, Germany
Christian.Prehofer@icn.siemens.de

Abstract

Functional logic languages with a sound and complete operational semantics are mainly based on an inference rule called narrowing. Narrowing extends functional evaluation by goal solving capabilities as in logic programming. 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 independent 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. We show soundness and completeness of our strategy with respect to LNT reductions, a particular form of higher-order reductions defined via definitional trees. A general completeness result is only provided for terminating rewrite systems due to the lack of an overall theory of higher-order reduction which is outside the scope of this paper.

1 Introduction

Functional logic languages (Hanus, 1994) with a sound and complete operational semantics are mainly based on narrowing. Narrowing, originally introduced in automated theorem proving (Slagle, 1974), 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.

[1] A preliminary short version of this paper appeared in the Proceedings of the Seventh International Conference on Rewriting Techniques and Applications (RTA'96), Springer LNCS 1103, pp. 138–152, 1996. This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-1.
[2] This work was mainly done during the author's stay at the Technical University of Munich.

Example 1.1
Consider the following rules defining the less-or-equal predicate on natural numbers which are represented by terms built from 0 and s:

$$\begin{array}{rcll} 0 & \leq & X & \rightarrow \ true \\ s(X) & \leq & 0 & \rightarrow \ false \\ s(X) & \leq & s(Y) & \rightarrow \ X \leq Y \end{array}$$

To solve the goal $s(X) \leq Y$, 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:

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

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* (Antoy *et al.*, 1994) 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 1.1 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 (Antoy *et al.*, 1994). Since needed narrowing is defined for *inductively sequential rewrite systems* (Antoy, 1992) (these are constructor-based rewrite systems with discriminating left-hand sides, i.e., typical functional programs), it can be efficiently implemented by pattern-matching and unification due to its local computation of a narrowing step (see, e.g., (Antoy, 1996; Hanus, 1995; Loogen *et al.*, 1993)).

On ground terms, needed narrowing falls back to the classical notion of "needed reduction" in the sense of Huet and Lévy (1991). Evaluation by needed narrowing has also some similarities to pattern matching and lazy evaluation in functional languages like Haskell (Hudak *et al.*, 1992) or Miranda (Peyton Jones, 1987). Note, however, that needed narrowing or needed reduction is a more powerful evaluation strategy than the simpler left-to-right pattern matching in current functional languages. For instance, consider the rules

$$\begin{array}{rcl} f(0,0) & \rightarrow & 0 \\ f(X,s(N)) & \rightarrow & 0 \end{array}$$

and a non-terminating function $\bot$. Since only the evaluation of the second argument of f is needed in order to apply a reduction rule, needed narrowing does not evaluate the first argument of the function call $f(\bot, s(0))$. Thus, needed narrowing reduces this expression to 0, in contrast to Miranda or Haskell which do not terminate on this function call. In general, needed narrowing/rewriting always computes a normal form if it exists (Antoy, 1992).

In this paper, we extend the needed narrowing strategy to higher-order functions and λ-terms as data structures (the idea to use λ-terms as data structures stems from λProlog (Nadathur & Miller, 1988)). In contrast to current functional languages, we permit free (existentially quantified) variables and λ-abstractions in expressions, and the latter also in left-hand sides of rewrite rules. This allows one to manipulate objects, such as formulas and programs, whose representation requires structures containing abstractions or bound variables. For instance, this has interesting applications in the verification and synthesis of software and hardware (Prehofer, 1995b; Prehofer, 1996; Prehofer, 1997). Furthermore, universally quantified goals are possible, as a goal $\forall x.s = \forall x.t$ is equivalent to $\lambda x.s = \lambda x.t$.

As a simple example of such higher-order functional logic programs, we define the differential of a function at some point.

Example 1.2
Consider the following rules defining a higher-order function *diff*, where $diff(F, X)$ computes the differential of F at X (we abbreviate $s(0)$ by 1):

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

With these rules, we can compute the differential of the double application of the function sin:

$$\begin{array}{lcl} \lambda x.diff(\lambda y.sin(sin(y)), x) & \to & \lambda x.cos(sin(x)) * diff(\lambda y.sin(y), x) \\ & \to & \lambda x.cos(sin(x)) * cos(x) * diff(\lambda y.y, x) \\ & \to & \lambda x.cos(sin(x)) * cos(x) * 1 \end{array}$$

Since we also allow free variables in expressions which are solved by narrowing, our calculus is able to synthesize new functions satisfying some goal. For instance, the equation

$$\lambda x.diff(\lambda y.sin(F(x, y)), x) = \lambda x.cos(x)$$

is solved by instantiating the higher-order variable F by the projection function $\lambda x, y.y$.

The interesting point in this example is the structure of the left-hand side of the rules. In order to apply a rule for *diff*, the first argument must always be evaluated to the form $\lambda y.v(\cdots)$ with $v \in \{y, sin, ln\}$, whereas the second argument can be arbitrary. In this sense, the first argument of *diff* is needed in contrast to the second. This property provides an efficient evaluation strategy similar to the first-order case. To summarize, the main contributions of this paper are as follows.

- 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. Furthermore, so-called flex-flex pairs do not have to be considered in this case, in contrast to general higher-order unification.

- Since our narrowing calculus LNT is oriented towards previous work on higher-order narrowing (Prehofer, 1997), we show that LNT coincides with needed narrowing in the first-order case. Note that steps of classic narrowing strategies (e.g., (Hullot, 1980; Slagle, 1974)) are defined as a variable instantiation followed by the reduction of some *subterm*, whereas more recent lazy narrowing strategies (Hölldobler, 1989; Ida & Nakahara, 1997; Martelli *et al.*, 1989; Middeldorp *et al.*, 1996; Nakahara *et al.*, 1995) manipulate equation systems in the style of Martelli and Montanari's unification algorithm (Martelli & Montanari, 1982) and always reduce outermost function symbols instead of subterms. Thus, we present, for the first time, a needed narrowing calculus in the Martelli/Montanari style and show its equivalence with the original formulation. This leads to a better understanding and a formal comparison of the different calculi. In contrast to (Ida & Nakahara, 1997; Middeldorp *et al.*, 1996), we integrate the pattern matching into the narrowing steps via definitional trees. As a consequence, our calculus is completely deterministic on ground terms, whereas (Ida & Nakahara, 1997; Middeldorp *et al.*, 1996) implement pattern matching by non-deterministic search.
- For the higher-order case, we show soundness and completeness with respect to LNT reductions, a particular form of higher-order reductions which we define via definitional trees. A general completeness result w.r.t. arbitrary reduction would be interesting but is outside the scope of this paper, since an overall theory of higher-order needed reduction has not been developed up to now. Nevertheless, we strongly conjecture that LNT reductions are in fact normalizing which is witnessed by the following facts:

  — LNT reductions are needed for reduction to a constructor normal form.
  — LNT reductions fall back to (first-order) needed reductions, which are known to be normalizing (Huet & Lévy, 1991), if the higher-order features are not used.

  For terminating higher-order inductively sequential rewrite systems, we show that LNT reductions compute a ground constructor normal form, if one exists. This implies a general completeness result for our strategy w.r.t. arbitrary reduction. Note that termination is also required for other higher-order narrowing calculi (Prehofer, 1997). In the context of functional logic languages, the definition of infinite data structures does not need non-terminating rules because infinite data structures can also be defined by terminating rules and the use of existential variables (see (Prehofer, 1997), Section 8.1.5). Nevertheless, a demand-driven evaluation strategy like ours is also useful for terminating rewrite systems since it evaluates subterms only when needed.
- We show that the calculus is optimal w.r.t. the solutions computed, i.e., no solution is produced twice. Since optimality of higher-order needed reductions is subject of current research, we show that LNT reductions are in fact needed for reduction to a constructor normal form. Thus, 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 narrow-

ing 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 (similar to λProlog (Nadathur & Miller, 1988) or Escher (Lloyd, 1994)), 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 (Avenhaus & Loría-Sáenz, 1994).

After recalling basic notions from the λ-calculus and term rewriting, we relate in Section 3 the original first-order needed narrowing calculus with the lazy narrowing calculus LNT in the style of Martelli and Montanari and show the equivalence of both calculi. Section 4 introduces higher-order inductively sequential rewrite systems and the extension of our calculus LNT to such programs is shown in Section 5. We proof soundness and completeness results in Section 6 and an optimality result in Section 7. Finally, Section 8 discusses criteria to avoid the sometimes operationally complex higher-order unification features of LNT.

2 Preliminaries

We briefly introduce the simply typed λ-calculus (see e.g. (Hindley & Seldin, 1986)). 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 $\rightarrow$. Note that $\rightarrow$ is right-associative, i.e., $\alpha \rightarrow \beta \rightarrow \gamma = \alpha \rightarrow (\beta \rightarrow \gamma)$. We assume for all types $\tau \in \mathcal{T}$ a set of variables V_τ and a set of (function) constants C_τ, where $V_\tau \cap V_{\tau'} = C_\tau \cap C_{\tau'} = \{\}$ for $\tau \neq \tau'$. If $f \in C_{\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau_0}$ where $\tau_0 \in \mathcal{T}_0$, then we call n the **arity** of f. 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.

Further, we often use s and t for terms and u, v, w for constants or bound variables.

The syntax for **λ-terms** is given by

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

Type judgments are written as $t : \tau$. The following inference rules inductively define the set of **simply typed λ-terms**.

$$\frac{x \in V_\tau}{x : \tau} \qquad \frac{c \in C_\tau}{c : \tau}$$

$$\frac{s : \tau \rightarrow \tau' \quad t : \tau}{(s\ t) : \tau'} \qquad \frac{x : \tau \quad s : \tau'}{(\lambda x.s) : \tau \rightarrow \tau'}$$

In the following, we only consider *simply typed* λ-terms (substitutions, equations etc). We write $\lambda x_1, x_2, \ldots, x_n.t$ for $\lambda x_1.(\lambda x_2. \ldots (\lambda x_n.t) \ldots)$. 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 $\lambda\overline{x_n}.s = \lambda x_1, \ldots, x_n.s$ 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. The **composition** $\varphi\sigma$ of two substitutions φ and σ is defined as $(\varphi\sigma)(t) = \varphi(\sigma(t))$. 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 (where $\{x \mapsto y\}t$ denotes the result of replacing every free occurrence of x in t by y):

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

The long $\beta\eta$-normal form (Nipkow, 1991) of a term t, denoted by $t\downarrow^\eta_\beta$, is the η-expanded form of the β-normal form of t. It is well known (Hindley & Seldin, 1986) that $s =_{\alpha\beta\eta} t$ iff $s\downarrow^\eta_\beta =_\alpha t\downarrow^\eta_\beta$. As long $\beta\eta$-normal forms exist for typed λ-terms, we will in general assume that terms are in long $\beta\eta$-normal form. For brevity, we may write variables in η-normal form, e.g., X instead of $\lambda\overline{x_n}.X(\overline{x_n})$. We assume that the transformation into long $\beta\eta$-normal form is an implicit operation, e.g., when applying a substitution to a term.

A substitution θ is in long $\beta\eta$-normal form if all terms in the image of θ are in long $\beta\eta$-normal form. The convention that α-equivalent terms are identified and that free and bound variables are kept disjoint (see also (Barendregt, 1984)) 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 $\theta X = \theta' X$ for all $X \in W$. The restriction of a substitution to a set of variables W is defined as $\theta_{|W} X = \theta X$ if $X \in W$ and $\theta_{|W} X = X$ otherwise. A substitution θ is **idempotent** iff $\theta = \theta\theta$. A substitution θ' is more general than θ over a set of variables W, written as $\theta' \leq_W \theta$, if $\theta =_W \sigma\theta'$ for some substitution σ. Two terms s and t are **unifiable** if there exists a substitution σ, also called a **unifier** for s and t, with $\sigma s = \sigma t$. A unifier σ for s and t is called **most general** if for each other unifier σ' for s and t there exists a substitution φ with $\sigma' = \varphi\sigma$.

We describe positions in λ-terms by sequences over natural numbers. The **subterm** of s at **position** p, written as $s|_p$, is defined as

- $s|_\epsilon = s$
- $v(\overline{t_m})|_{i.p} = t_i|_p$ if $1 \leq i \leq m$
- $\lambda\overline{x_m}.t|_{1.p} = (\lambda x_2, \ldots, x_m.t)|_p$
- undefined otherwise

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 (Miller, 1991). Examples of higher-order patterns are $\lambda x, y.F(x, y)$ and $\lambda x.f(G(\lambda z.x(z)))$, where the latter is at least third-order. Non-patterns are for instance $\lambda x, y.F(a, y)$ and $\lambda x.G(H(x))$.

A **rewrite rule** (Nipkow, 1991) is a pair $l \to r$ 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. Similarly to current functional languages, we consider only (first-order) constructor terms (and not arbitrary normal forms) as values which can be observed by the user. To simplify our calculi, we consider only the evaluation of expressions to 0-ary constructor constants, i.e., an evaluation of a term t has the form $t \to \cdots \to c$ where c is a 0-ary constructor. In the presence of free variables, it might be necessary to compute values for these free variables such that the instantiated expression is reducible. This can be done by narrowing which will be precisely defined in the following sections.

This notion of evaluation to 0-ary constructors is general enough to cover a variety of different evaluation tasks occurring in functional logic languages. For instance, if we are interested in the reduction of an expression to a constructor term c, we add the rule $f(c) \to ok$, where f and ok are new symbols. Then, t is reducible to c iff $f(t)$ is reducible to the constructor ok. For instance, the equation shown in Example 1.2 above can be solved by adding the rule

$$f(\lambda x.cos(x)) \to ok$$

and evaluating the term $f(\lambda x.\mathit{diff}(\lambda y.sin(F(x, y)), x))$. If this term is evaluated (by narrowing) to ok, then the computed binding for the free variable F is a solution of the equation.

When we are interested to evaluate expressions to (first-order) ground constructor terms, we could add new symbols $true$, $\wedge$, and $ground$ (more precisely, one $ground$ symbol for each base type), together with the following rules, where $\wedge$ is assumed to be a right-associative infix symbol, and c is a constructor of arity 0 in the first rule and arity $n > 0$ in the second rule. Then, t is reducible to a first-order ground

constructor term s if $ground(t)$ is reducible to $true$.

$$\begin{array}{rcl} ground(c) & \to & true \\ ground(c(X_1,\ldots,X_n)) & \to & ground(X_1) \wedge \cdots \wedge ground(X_n) \\ true \wedge X & \to & X \end{array}$$

Finally, we can use a similar technique to cover the equation solving capabilities of current functional logic languages with a lazy operational semantics, like BABEL (Moreno-Navarro & Rodríguez-Artalejo, 1992) or K-LEAF (Giovannetti *et al.*, 1991), since the strict equality "$\approx$" ($t_1 \approx t_2$ holds if t_1 and t_2 are reducible to a same ground constructor term) can be defined as a binary operation by a set of orthogonal rewrite rules. For this purpose, consider a family of (infix) symbols $\approx$ (one operation for each base type) defined by the following rules (as above, c is a constructor of arity 0 in the first rule and arity $n > 0$ in the second rule):

$$\begin{array}{rcl} c \approx c & \to & true \\ c(X_1,\ldots,X_n) \approx c(X_1,\ldots,X_n) & \to & (X_1 \approx Y_1) \wedge \cdots \wedge (X_n \approx Y_n) \\ true \wedge X & \to & X \end{array}$$

A strict equation is **valid** if it can be rewritten to $true$ with these rules (see (Antoy *et al.*, 1994; Giovannetti *et al.*, 1991; Moreno-Navarro & Rodríguez-Artalejo, 1992) for more details about strict equality, where in (Moreno-Navarro & Rodríguez-Artalejo, 1992) additional rules for disequalities are added to derive the result $false$ for an equation). Note that normal forms may not exist in general due to non-terminating rewrite rules. As a particular case, a first-order ground constructor normal form of an expression t can be computed by solving the equation $t \approx X$ where X is a free variable, since $t \approx s$ is reducible to $true$ iff t and s are reducible to a same ground constructor term (see (Antoy *et al.*, 1994), Proposition 1).

This interpretation of equality in goals is also taken in functional logic languages with higher-order features (e.g., (González-Moreno *et al.*, 1992) proposes a slightly extended definition without explicit rewrite rules). Although this explicit definition of strict equality only supports equalities between first-order terms, higher-order terms (i.e., functions) can occur in data structures as long as these data structures are not checked for equality. For instance, it is possible in our framework to compute the length of a list of functions since the functional elements do not occur in the result of this computation. Furthermore, higher-order terms may occur as arguments in the left-hand sides of rewrite rules so that new functions may be synthesized by our higher-order narrowing strategy (in contrast to current purely functional languages), as shown in Example 1.2.

We often consider a **goal** as an expression of Boolean type that should be reduced to the constant $true$. A substitution σ is a **solution** of a goal G iff $\sigma(G)$ can be rewritten to $true$. Note, however, that we will present our soundness and completeness results for more general terms rather than goals.

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'.

3 First-Order Definitional Trees

Definitional trees are introduced in (Antoy, 1992) to define efficient normalization strategies for (first-order) term rewriting. The idea is to represent all rules for a defined symbol in a tree and to control the selection of the next redex by this tree. This technique is extended to narrowing in (Antoy *et al.*, 1994) where it is shown that a narrowing strategy based on definitional trees is optimal in the length of narrowing derivations and the number of computed solutions. 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, which is more appropriate with regard to the extension to the higher-order case. Thus, we assume in this section that all terms are **first-order**, i.e., λ-abstractions, functional variables and partial applications (i.e., applications to less arguments than the arity of a symbol, which correspond to λ-abstractions after η-expansion) do not occur.

A traditional narrowing step (Hanus, 1994) is defined by computing a most general unifier of a subterm of the current term and the left-hand side of a rewrite rule, applying this unifier to the current term and replacing the subterm by the instantiated right-hand side of the rule. More precisely, 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 $t \leadsto_{\sigma_{|\mathcal{FV}(t)}} t'$ (since we are interested only in the instantiation of the goal variables, we omit the bindings of the other local variables in the narrowing steps). 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 (Hanus, 1994) for a survey). A **narrowing strategy** determines the position where the next narrowing step should be applied. As shown in (Antoy *et al.*, 1994), an optimal narrowing strategy can be obtained by dropping the requirement for computing a most general unifier in each narrowing step and controlling the instantiation of variables and selection of narrowing positions by a data structure, called definitional tree. For instance, consider the rules of Example 1.1 together with the rule

$$f(0) \to 0$$

and the goal $X \leq f(Y)$. The second argument $f(Y)$ needs only to be evaluated if

X is bound to a term of the form $s(\cdots)$. Therefore, needed narrowing instantiates X either to 0 (and reduces the goal to *true* without evaluating $f(Y)$) or to $s(Z)$. In the latter case, $f(Y)$ must be evaluated. However, the resulting needed narrowing step

$$X \leq f(Y) \;\leadsto_{\{X \mapsto s(Z), Y \mapsto 0\}}\; s(Z) \leq 0$$

is not a traditional narrowing step with a most general unifier. On the other hand, traditional lazy narrowing (Moreno-Navarro & Rodríguez-Artalejo, 1992) narrows the second argument $f(Y)$ since it is a demanded argument and binds in the subsequent narrowing step X, i.e., the derivation

$$X \leq f(Y) \;\leadsto_{\{Y \mapsto 0\}}\; X \leq 0 \;\leadsto_{\{X \mapsto 0\}}\; true$$

is a lazy narrowing derivation in the sense of (Moreno-Navarro & Rodríguez-Artalejo, 1992) which unnecessarily evaluates the subterm $f(Y)$. Thus, this derivation is not optimal.

To provide a precise definition of this needed narrowing strategy, we first recall the notion of a definitional tree (the following definition slightly differs from Antoy's (1992) original definition since we ignore *exempt* nodes). $\mathcal{T}$ is a **definitional tree** with pattern π (where π has the form $f(\overline{t_n})$ with f defined symbol and $\overline{t_n}$ constructor terms and each variable in π occurs only once) iff its depth is finite and one of the following cases holds:

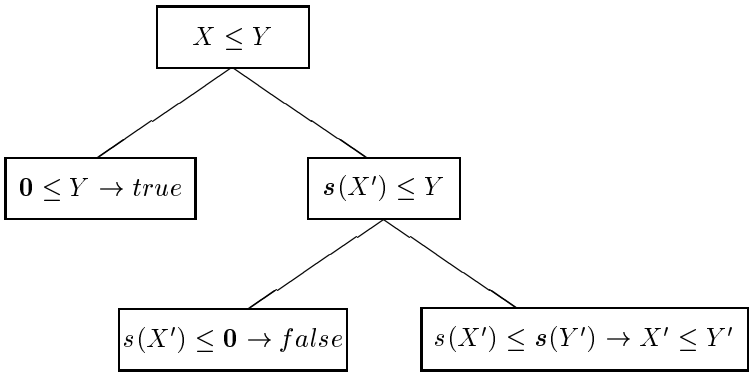
$\mathcal{T} = rule(l \to r)$, where $l \to r$ is a variant of a rule in $\mathcal{R}$ such that $l = \pi$.

$\mathcal{T} = branch(\pi, o, \overline{\mathcal{T}_k})$, where o is an occurrence of a variable in π, $\overline{c_k}$ are pairwise 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.

We denote by $pat(\mathcal{T})$ the **pattern** of the definitional tree $\mathcal{T}$. 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 $l \to r \in \mathcal{R}$ with $l = f(\overline{t_n})$, there is a node $rule(l' \to r')$ in $\mathcal{T}$ with $l \to r$ variant of $l' \to r'$. For instance, the rules in Example 1.1 can be represented by the following definitional tree:

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

This tree can be illustrated by the following picture:



A definitional tree starts always with the most general pattern $f(\overline{X_n})$ for a defined symbol f 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. The term rewriting system $\mathcal{R}$ is called inductively sequential if each function defined by $\mathcal{R}$ is inductively sequential. Thus, inductively sequential rewrite rules are a subclass of constructor-based orthogonal rewrite systems which are appropriate to reflect current (first-order) functional languages.

3.1 Narrowing with Definitional Trees

A definitional tree defines a strategy to evaluate functions by applying narrowing steps. Since there may exist more than one definitional tree for a given function, we assume in the following that a definitional tree is fixed for each function, i.e., we talk about *the* definitional tree of a function. Note that the soundness and completeness results are independent of the chosen definitional trees. Different trees may only influence the failure behavior of needed narrowing. An algorithm for the construction of definitional trees can be found in (Hanus, 1997).

To narrow a term t, we consider the definitional tree $\mathcal{T}$ of the outermost function symbol of t (note that, since we consider only evaluations to 0-ary constructors, the outermost symbol of a term to be evaluated is always a defined function).

$\mathcal{T} = rule(l \rightarrow r)$**:** Apply rule $l \rightarrow r$ to t (note that t is always an instance of l).
$\mathcal{T} = branch(\pi, o, \overline{\mathcal{T}_k})$**:** Consider the subterm $t|_o$.

1. If $t|_o$ has a function symbol at the top, we narrow this subterm (to a term without a defined symbol at the top) by recursively applying our strategy to $t|_o$.
2. If $t|_o$ has a constructor symbol at the top, we narrow t with $\mathcal{T}_j$, where the pattern of $\mathcal{T}_j$ matches t, otherwise (if no pattern matches) we fail.
3. If $t|_o$ is a variable, we non-deterministically select a subtree $\mathcal{T}_j$, unify t with

the pattern of $\mathcal{T}_j$ (i.e., $t|_o$ is instantiated to the constructor of the pattern of $\mathcal{T}_j$ at position o), and narrow this instance of t with $\mathcal{T}_j$.

For instance, consider the rules of Example 1.1 and the goal $0 \leq f(Z)$. Since the definitional tree of $\leq$ is a branch on the first argument and the actual first argument is the constructor 0, we proceed the evaluation of $0 \leq f(Z)$ with the subtree $rule(0 \leq Y \to true)$, which is the only subtree whose pattern matches the goal. Since this subtree is a rule, we apply the rule (without evaluating $f(Z)$) and obtain the result *true*.

This strategy, called **needed narrowing** (Antoy *et al.*, 1994), 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.

A formal description of this strategy in terms of an inference system is shown in Figure 1. If we want to narrow the operation-rooted term t to some constructor, we apply inference steps to the term t until we obtain the constructor or we fail.[3] An $\mathcal{E}val$-**goal** is any sequence obtained from such an initial goal by applying inference steps of this calculus. The inference rule **Initial** decorates the initial term with the appropriate definitional tree. If this tree is a rule, then the inference rule **Apply** applies an instance of this rule to the current term. **Select** selects the appropriate subtree of the current definitional tree, and **Instantiate** non-deterministically selects a subtree of the current definitional tree and instantiates the variable at the current position to the appropriate pattern. The rule **Eval Subterm** initiates the evaluation of the subterm at the current position by creating a new $\mathcal{E}val$-goal for this subterm. As in proof procedures for logic programming, we assume that we take a definitional tree with fresh variables in each such evaluation step. This implies that all computed substitutions are idempotent (we will implicitly assume this property in the following). If a rewrite rule has been applied to this subterm (by inference rule **Apply**), the rewritten subterm is inserted at the current position by the inference rule **Replace Subterm**.

Example 3.1
Consider the rules of Example 1.1 and the initial goal $s(X) \leq Y$. The following derivation shows the computation of the answer $\{X \mapsto 0, Y \mapsto s(Y_2)\}$ in the needed narrowing calculus. In contrast to traditional narrowing steps, where a subterm is directly unified with the left-hand side of some rewrite rule, the derivation steps in the needed narrowing calculus explicitly show the selection of the subterm and the instantiation of the goal variables.

$$s(X) \leq Y$$
$$\Rightarrow_{Initial}$$
$$\mathcal{E}val(s(X) \leq Y, branch(X_1 \leq Y_1, 1, rule(\cdots), branch(s(X_2) \leq Y_1, \ldots)))$$

[3] This description of needed narrowing is slightly different than in (Antoy *et al.*, 1994) but more appropriate for the subsequent proofs. In (Antoy *et al.*, 1994), the term to be narrowed is always traversed from the root to the narrowing position in each narrowing step, whereas the traversal is represented here by a sequence of $\mathcal{E}val$-goals.

Initial

$$t \Rightarrow^{\{\}} \mathcal{E}val(t, \mathcal{T}) \quad \text{if } t = f(\overline{t_n}) \text{ and } \mathcal{T} \text{ is the definitional tree of } f$$

Apply

$$\mathcal{E}val(t, rule(l \to r)), G \Rightarrow^{\{\}} \sigma(r), G \quad \text{if } \sigma(l) = t$$

Select

$$\mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k})), G \Rightarrow^{\{\}} \mathcal{E}val(t, \mathcal{T}_i), G \quad \text{if } t|_o = c(\overline{t_n}) \text{ and } pat(\mathcal{T}_i)|_o = c(\overline{X_n})$$

Instantiate

$$\mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k})), G \Rightarrow^{\sigma} \sigma(\mathcal{E}val(t, \mathcal{T}_i), G) \quad \text{if } t|_o = X \text{ (variable) and } \sigma = \{X \mapsto pat(\mathcal{T}_i)|_o\}$$

Eval Subterm

$$\mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k})), G \Rightarrow^{\{\}} \mathcal{E}val(t|_o, \mathcal{T}), \mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k})), G \quad \text{if } t|_o = f(\overline{t_n}) \text{ and } \mathcal{T} \text{ is the definitional tree of } f$$

Replace Subterm

$$t', \mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k})), G \Rightarrow^{\{\}} \mathcal{E}val(t[t']_o, branch(\pi, o, \overline{\mathcal{T}_k})), G \quad \text{if } t' \neq \mathcal{E}val(\ldots, \ldots)$$

Fig. 1. Calculus for needed narrowing

$$\Rightarrow_{Select}$$
$$\mathcal{E}val(s(X) \leq Y, branch(s(X_2) \leq Y_1, 2, rule(s(X_2) \leq 0 \to false), rule(s(X_2) \leq s(Y_2) \to X_2 \leq Y_2)))$$
$$\Rightarrow^{\{Y \mapsto s(Y_2)\}}_{Instantiate}$$
$$\mathcal{E}val(s(X) \leq s(Y_2), rule(s(X_2) \leq s(Y_2) \to X_2 \leq Y_2))$$
$$\Rightarrow_{Apply}$$
$$X \leq Y_2$$
$$\Rightarrow_{Initial}$$
$$\mathcal{E}val(X \leq Y_2, branch(X_3 \leq Y_3, 1, rule(0 \leq Y_3 \to true), branch(s(X_4) \leq Y_3, \ldots)))$$
$$\Rightarrow^{\{X \mapsto 0\}}_{Instantiate}$$
$$\mathcal{E}val(0 \leq Y_2, rule(0 \leq Y_3 \to true)) \Rightarrow_{Apply} true$$

3.2 Narrowing with Case Expressions

In order to extend this strategy to higher-order functions, another representation is useful since an explicit representation of the structure of definitional trees in the rewrite rules provides more explicit control which leads to a simpler calculus. Also, it is shown in (Prehofer, 1994) that the direct application of narrowing steps to inner subterms should be avoided in the presence of λ-bound variables. This new representation will lead to an interesting comparison of needed rewrite sequences and leftmost outermost rewriting with case expressions.

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 by 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. The variables $\overline{X_{n_i}}$ are called **pattern variables** and are local variables which occur only in the corresponding subexpression $\mathcal{X}_i$.

Using case expressions, each inductively sequential function symbol can be defined by exactly one rewrite rule, where the left-hand side consists always of the function symbol applied to different variables and the right-hand side is a representation of the corresponding definitional tree by case expressions. For instance, the rules for the function $\leq$ defined in Example 1.1 are represented by the following rewrite rule:

$$\begin{array}{lllllllll} X \leq Y & \rightarrow & case\ X\ of & 0 & : & true, & & & \\ & & & s(X_1) & : & case\ Y\ of & 0 & : & false, \\ & & & & & & s(Y_1) & : & X_1 \leq Y_1 \end{array}$$

Although this is not a rewrite rule in the traditional sense (due to the fresh pattern variables), we will provide a unique operational reading by specifying a particular semantics to case expressions. A case expression can be considered as a function symbol whose semantics is defined by a set of rewrite rules. For instance, the last case expression is considered as a function of arity 5 ("$case(X, 0, true, s(X_1), case(Y, \ldots))$", but we still use the mixfix notation in this paper) together with the following rewrite rules (where "$_$" denotes an arbitrary anonymous variable):

$$\begin{array}{lll} case\ 0\ \ of\ 0 : T, & _ : _ & \rightarrow\ T \\ case\ s(X)\ of\ _ : _, & s(X) : T & \rightarrow\ T \end{array}$$

Although the left-hand side of the last rule is not linear, the multiple occurrence of the variable X will be only used to pass subterms from one place to another and not for comparing terms. This will become clear from the special use of case expressions (see below).

We formalize the mapping of a definitional tree $\mathcal{T}$ into a term with case expressions by the use of the translation function $\mathcal{C}ase(\mathcal{T})$:

$$\begin{array}{lcllll} \mathcal{C}ase(rule(l \rightarrow r)) & = & r & & & \\ \mathcal{C}ase(branch(\pi, o, \overline{\mathcal{T}_k})) & = & case\ \ \pi|_o\ \ of & pat(\mathcal{T}_1)|_o & : & \mathcal{C}ase(\mathcal{T}_1), \\ & & & \vdots & & \\ & & & pat(\mathcal{T}_k)|_o & : & \mathcal{C}ase(\mathcal{T}_k) \end{array}$$

If $\mathcal{T}$ is the definitional tree with pattern $f(\overline{X_n})$ of the n-ary function f, then

$$f(\overline{X_n}) \ \rightarrow\ \mathcal{C}ase(\mathcal{T})$$

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 $2n+1$ with the following n rewrite rules:[4]

$$\begin{array}{lcl} case\ p_1\ of\ p_1 : X, \ldots, _ : _ & \to & X \\ \vdots & & \\ case\ p_n\ of\ _ : _, \ldots, p_n : X & \to & X \end{array}$$

If we apply a narrowing step to an expression of the form $case\ t\ of\ p_1 : t_1, \ldots, p_n : t_n$, there are two principal possibilities:

1. If t is a variable, we can apply any of the n defining rules for *case*, i.e., there are n possible narrowing steps

   $$case\ t\ of\ p_1 : t_1, \ldots, p_n : t_n \ \leadsto_\sigma \ \sigma(t_i)$$

   with $\sigma = \{t \mapsto p_i\}$ $(i \in \{1, \ldots, n\})$. This corresponds to an instantiation step in the needed narrowing calculus.
2. If t is a constructor-rooted term $c(\overline{s_k})$ and $p_i = c(\overline{X_k})$ for some $i \in \{1, \ldots, n\}$, then

   $$case\ t\ of\ p_1 : t_1, \ldots, p_n : t_n \ \leadsto_{\{\}} \ \sigma(t_i)$$

   for $\sigma = \{\overline{X_k \mapsto s_k}\}$. We write $\leadsto_{\{\}}$ instead of $\leadsto_\sigma$ since we are interested only in the instantiation of goal variables and the pattern variables occur only locally in the expression t_i. This step corresponds to a selection step in the needed narrowing calculus.

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 $\mathcal{R}_{case}$ the additional *case* rewrite rules. We will show a strong correspondence between derivations in the needed narrowing calculus and leftmost-outermost narrowing derivations. **Leftmost-outermost narrowing** means that the selected subterm is the leftmost-outermost one among all possible narrowing positions, where we call a position p **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 $i < j$.

Example 3.2

Consider again the rules of Example 1.1 and the initial goal $s(X) \leq Y$. The following derivation is a sequence of leftmost-outermost narrowing steps to compute the answer $\{X \mapsto 0, Y \mapsto s(Y_1)\}$. The applied rewrite rules are the single rule for

[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.

$\leq$ together with the rewrite rules for case expressions as shown above.

$$
\begin{array}{ll}
& s(X) \leq Y \\
\leadsto_{\{\}} & \mathit{case}\ s(X)\ \mathit{of}\ 0 : \mathit{true},\ s(X_1) : (\mathit{case}\ Y\ \mathit{of}\ 0 : \mathit{false},\ s(Y_1) : X_1 \leq Y_1) \\
\leadsto_{\{\}} & \mathit{case}\ Y\ \mathit{of}\ 0 : \mathit{false},\ s(Y_1) : X \leq Y_1 \\
\leadsto_{\{Y \mapsto s(Y_1)\}} & X \leq Y_1 \\
\leadsto_{\{\}} & \mathit{case}\ X\ \mathit{of}\ 0 : \mathit{true},\ s(X_2) : (\mathit{case}\ Y_1\ \mathit{of}\ 0 : \mathit{false},\ s(Y_2) : X_2 \leq Y_2) \\
\leadsto_{\{X \mapsto 0\}} & \mathit{true}
\end{array}
$$

The equivalence of needed narrowing w.r.t. $\mathcal{R}$ and leftmost-outermost narrowing w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$ is stated in the following theorem. The proof can be found in the appendix.

Theorem 3.3

Let t be a term and c be a 0-ary constructor. For each needed narrowing derivation $t \leadsto_\sigma^* c$ w.r.t. $\mathcal{R}$ there exists a leftmost-outermost narrowing derivation $t \leadsto_\sigma^* c$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$, 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. This is often different from the leftmost-outermost narrowing position. For instance, the term $s(X) \leq X + X$ has the outermost function symbol $\leq$ which is different from the symbol $+$ at the leftmost-outermost narrowing position. In order to obtain a calculus which always applies steps at the "top level", we transform a leftmost-outermost narrowing derivation w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$ 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 2. 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 evaluating expressions to a 0-ary constructor c, we assume that the **initial goal** $\mathcal{I}_c(t)$ for a term t has always the form $\mathit{case}\ t\ \mathit{of}\ c : c \to^? T$ where T is a variable not occurring in t. We use this representation in order to provide a calculus with fewer inference rules. Note that T will be bound to c if such a goal can be reduced to the empty goal system.

Example 3.4

The following LNT-derivation corresponds to the leftmost-outermost narrowing de-

Bind

$$e \to^? Z, G \;\Rightarrow^{\{\}}\; \sigma(G)$$
where $\sigma = \{Z \mapsto e\}$ and e is not a case term

Case Select

$$case\ c(\overline{t_n})\ of\ \overline{p_k : \mathcal{X}_k} \to^? Z, G \;\Rightarrow^{\{\}}\; \sigma(\mathcal{X}_i) \to^? Z,\ G$$
if $p_i = c(\overline{X_n})$ and $\sigma = \{\overline{X_n \mapsto t_n}\}$

Case Guess

$$case\ X\ of\ \overline{p_k : \mathcal{X}_k} \to^? Z, G \;\Rightarrow^{\sigma}\; \sigma(\mathcal{X}_i) \to^? Z,\ \sigma(G)$$
where $\sigma = \{X \mapsto p_i\}$

Case Eval

$$case\ f(\overline{t_n})\ of\ \overline{p_k : \mathcal{X}_k} \to^? Z, G \;\Rightarrow^{\{\}}\; \sigma(\mathcal{X}) \to^? X,\ case\ X\ of\ \overline{p_k : \mathcal{X}_k} \to^? Z, G$$
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. 2. Calculus LNT for lazy narrowing with definitional trees in the first-order case

rivation shown in Example 3.2.

$$case\ s(X) \leq Y\ of\ true : true \to^? T \qquad (= \mathcal{I}_{true}(s(X) \leq Y))$$

$$\Rightarrow_{Case\ Eval}$$

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

$$\Rightarrow_{Case\ Select}$$

$$case\ Y\ of\ 0 : false,\ s(Y_1) : X \leq Y_1 \to^? Z,\ case\ Z\ of\ true : true \to^? T$$

$$\Rightarrow^{\{Y \mapsto s(Y_1)\}}_{Case\ Guess}$$

$$X \leq Y_1 \to^? Z,\ case\ Z\ of\ true : true \to^? T$$

$$\Rightarrow_{Bind}$$

$$case\ X \leq Y_1\ of\ true : true \to^? T$$

$$\Rightarrow_{Case\ Eval}$$

$$case\ X\ of\ 0 : true,\ s(X_2) : (case\ Y_1\ of\ 0 : false,\ s(Y_2) : X_2 \leq Y_2) \to^? Z',$$
$$case\ Z'\ of\ true : true \to^? T$$

$$\Rightarrow^{\{X \mapsto 0\}}_{Case\ Guess}$$

$$true \to^? Z',\ case\ Z'\ of\ true : true \to^? T$$

$$\Rightarrow_{Bind}$$

$$case\ true\ of\ true : true \to^? T$$

$$\Rightarrow_{Case\ Select} \quad true \to^? T \quad \Rightarrow_{Bind} \quad \{\}$$

The equivalence of leftmost-outermost narrowing and the lazy narrowing calculus LNT is stated in the following theorem. The proof can be found in the appendix.

Theorem 3.5
Let t be a term, c a 0-ary constructor, and X a fresh variable. For each leftmost-outermost narrowing derivation $t \leadsto^*_\sigma c$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$ there exists a LNT-derivation $case\ t\ of\ c : c \to^? X \overset{*}{\Rightarrow}^\sigma c \to^? X$ w.r.t. $\mathcal{R}'$, and vice versa.

Theorems 3.3 and 3.5 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 for the application of definitional trees 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 structures, e.g., higher-order terms with bound variables in the left-hand sides. For instance, we permit higher-order rules like in Example 1.2. In contrast to the original definition of needed narrowing in the first-order case, we provide a definition of higher-order definitional trees in terms of case expressions. The relationship of the original definitional trees and case expressions was extensively discussed in the previous section.

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

Definition 4.1
$\mathcal{T}$ is a **higher-order definitional tree (*hdt*)** iff its depth is finite and one of the following cases holds:

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

where p 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 *hdt*s $\overline{\mathcal{T}_n}$ must be pairwise non-unifiable.

When *hdt*s are used to represent the rules of a function, further restrictions on the occurrences of free variables will be made (see below). We write *hdt*s as $p : \mathcal{X}$, 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 $\mathcal{T} = p : \mathcal{X}$ is defined as $sel(\mathcal{T}) = p$. For a subtree $\mathcal{T}'$ 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 $pat_{\mathcal{T}}(\mathcal{T}')$:

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

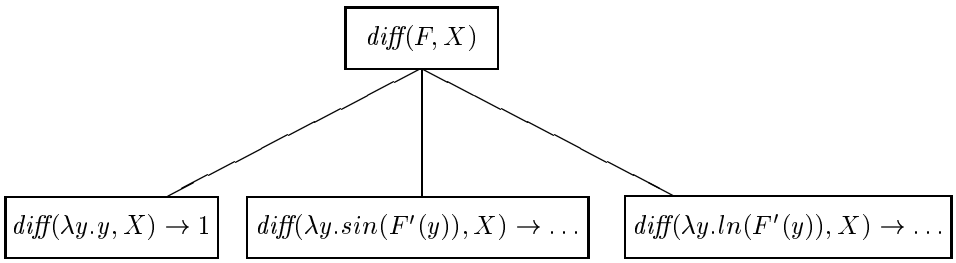
Each branch variable must belong to the pattern of this subtree, i.e., for each subtree $\mathcal{T}' = p : case\ X\ of\ \overline{\mathcal{T}_n}$ in a tree $\mathcal{T}$, X is a free variable of $pat_{\mathcal{T}}(\mathcal{T}')$. Furthermore, each leaf $\mathcal{T}' = p : rhs$ of a *hdt* $\mathcal{T}$ 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 where each defined symbol has a *hdt* is called **inductively sequential**.

For instance, the rules for *diff* in Example 1.2 have the *hdt*

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

For presenting definitional trees graphically, it is convenient to write $pat_{\mathcal{T}}(\mathcal{T}')$ for each subtree $\mathcal{T}'$. Thus we draw the tree for *diff* as:



A *hdt* can also have a more complicated branching structure, as shown by the definitional tree for Example 1.1 in Section 3 which is a special case of a *hdt*.

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 is also applied to study optimal reductions in (Oostrom, 1996), allows to find redices as in the first-order case, and furthermore simplifies narrowing. For instance, flex-flex pairs (equations between non-rigid terms) do not arise here, in contrast to the full higher-order case (Prehofer, 1995a; Prehofer, 1997). Consider an example for some non-overlapping rewrite rules which do not have a *hdt*:

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

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.

Bind

$$e \to^? Z, G \quad \Rightarrow^{\{\}} \quad \sigma(G)$$
where $\sigma = \{Z \mapsto e\}$ and e is not a case term

Case Select

$$\lambda\overline{x_k}.case\ \lambda\overline{y_l}.v(\overline{t_m})\ of\ \overline{p_n : \mathcal{X}_n} \to^? Z, G \quad \Rightarrow^{\{\}} \quad \lambda\overline{x_k}.\sigma(\mathcal{X}_i) \to^? Z, G$$
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

$$\lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{t_m})\ of\ \overline{p_n : \mathcal{X}_n} \to^? Z, G \quad \Rightarrow^{\sigma} \quad \sigma(\lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{t_m})\ of\ \overline{p_n : \mathcal{X}_n} \to^? Z, G)$$
if $p_i = \lambda\overline{y_l}.c(\overline{X_r(\overline{x_k}, \overline{y_l})})$ and $\sigma = \{X \mapsto \lambda\overline{x_m}.c(\overline{H_r(\overline{x_m})})\}$

Function Guess

$$\lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{t_m})\ of\ \overline{p_n : \mathcal{X}_n} \to^? Z, G \quad \Rightarrow^{\sigma} \quad \sigma(\lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{t_m})\ of\ \overline{p_n : \mathcal{X}_n} \to^? Z, G)$$
if $\lambda\overline{x_k}, \overline{y_l}.X(\overline{t_m})$ is not a higher-order pattern,
$\sigma = \{X \mapsto \lambda\overline{x_m}.f(\overline{H_r(\overline{x_m})})\}$, and f is a defined function

Projection

$$\lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{t_m})\ of\ \overline{p_n : \mathcal{X}_n} \to^? Z, G \quad \Rightarrow^{\sigma} \quad \sigma(\lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{t_m})\ of\ \overline{p_n : \mathcal{X}_n} \to^? Z, G)$$
where $\sigma = \{X \mapsto \lambda\overline{x_m}.x_i(\overline{H_r(\overline{x_m})})\}$

Case Eval

$$\lambda\overline{x_k}.case\ \lambda\overline{y_l}.f(\overline{t_m})\ of\ \overline{p_n : \mathcal{X}_n} \to^? Z, G \quad \Rightarrow^{\{\}} \quad \lambda\overline{x_k}, \overline{y_l}.\sigma(\mathcal{X}) \to^? X,\ \lambda\overline{x_k}.case\ \lambda\overline{y_l}.X(\overline{x_k}, \overline{y_l})\ of\ \overline{p_n : \mathcal{X}_n} \to^? Z, G$$
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. 3. System LNT for needed narrowing in the higher-order case

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, Function Guess and Projection rules have been added in Figure 3. 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. For all rules, we assume that newly introduced variables are fresh, as in the first-order case.

The **Imitation** and **Projection** rules are taken from higher-order unification and compute a partial binding for some variable. For instance, if the goal has the

form

$$case\ \lambda x.F(x)\ of\ \lambda y.sin(F'(y)) : \cdots\ ,$$

then the Imitation rule substitutes the higher-order variable F by the function $\lambda x.sin(H(x))$, which "imitates" the top function sin in the pattern of the case expression. Thus, the Imitation rule derives the (β-reduced) new goal

$$case\ \lambda x.sin(H(x))\ of\ \lambda y.sin(F'(y)) : \cdots$$

to which only Case Select is applicable in the following step. If we consider the goal

$$case\ \lambda x.F(x)\ of\ \lambda y.y : 1 \to^? Z\ ,$$

then we can apply the Projection rule to substitute the higher-order variable F by the projection function $\lambda x.x$ (this is the only possible projection function in this case since the function $\lambda x.F(x)$ has only one argument) and we obtain the (β-reduced) new goal $case\ \lambda x.x\ of\ \lambda y.y : 1 \to^? Z$ which can be further evaluated by Case Select.

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 instance, the goal

$$case\ F(0)\ of\ true : \cdots$$

can be derived with the Function Guess rule by replacing the higher-order variable F with a function that takes a number and produces a Boolean value as the result. Hence, if $\leq$ is the function defined in Example 1.1, an application of Function Guess can replace F with $\lambda x.H_1(x) \leq H_2(x)$ and derives the new goal

$$case\ H_1(0) \leq H_2(0)\ of\ true : \cdots$$

which can be further evaluated by applying the Case Eval rule.

Notice that for goals where only higher-order patterns occur, there is no choice between Projection and Imitation for matching with a single case branch. (For different branches, clearly Projection or Imitation can be used.) Furthermore, Function Guess does not apply. This special case is refined later in Section 8.

In contrast to the first-order case, locally bound variables may occur and must be correctly handled during unification. Therefore, all rewrite rules applied by **Case Eval** must be lifted into its correct binding context which is necessary to compute solutions for function variables with locally bound variables (see also (Prehofer, 1994)). Since free variables cannot be instantiated to bound variables, it is necessary to add the variables bound in the current context to all free variables of a rewrite rule so that they can be later instantiated by projection functions to identify them with the appropriate bound variables. A more technical example showing the need for lifting is provided in Example 5.2.

For a sequence $\Rightarrow^{\theta_1} \cdots \Rightarrow^{\theta_n}$ of LNT steps, we write $\stackrel{*}{\Rightarrow}^{\theta}$, where $\theta = \theta_n \cdots \theta_1$. As 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 (Prehofer, 1995a).

As in the first-order case, we consider only reductions to a dedicated 0-ary constructor c. Hence we assume that solving a goal $t \to^? c$ is initiated with the **initial goal** $\mathcal{I}_c(t) = case\ t\ of\ c : c \to^? X$.

Example 5.1
As an example for the computation in our calculus LNT, 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* (see Example 1.2) and the *hdt* for the function $*$:

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

As discussed in Section 2, we add the rule $f(\lambda x.cos(x)) \to ok$ (for which a *hdt* exists) and solve the goal $\mathcal{I}_{ok}(f(\lambda x.diff(\lambda y.sin(F(x,y)),x)))$. Since each computation step only affects the two leftmost goals, we often omit the others. Furthermore, we sometimes write expressions in η-normal form, e.g., *cos* instead of $\lambda x.cos(x)$.

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

$\Rightarrow_{Case\ Eval}$ (Apply the rule for f)

$case\ \lambda x.diff(\lambda y.sin(F(x,y)),x)\ of\ cos : ok \to^? X_2,\ \ case\ X_2\ of\ ok : ok \to^? X_1$

$\Rightarrow_{Case\ Eval}$ (Apply the rule for *diff* after lifting it with x)

$\lambda x.case\ \lambda y.sin(F(x,y))\ of\ \ldots,$
$\qquad\qquad \lambda y.sin(G(x,y)) : cos(G(x,x)) * diff(\lambda y.G(x,y),x),$
$\qquad\qquad \ldots \ \to^? X_3,$
$\qquad\qquad case\ X_3\ of\ cos : ok \to^? X_2,\ \ case\ X_2\ of\ ok : ok \to^? X_1$

$\Rightarrow_{Case\ Select}$ (Only case selection is possible)

$\lambda x.cos(F(x,x)) * diff(\lambda y.F(x,y),x) \to^? X_3,\ \ case\ X_3\ of\ cos : ok \to^? X_2, \ldots$

$\Rightarrow_{Bind}$ (First case expression is evaluated, bind result to X_3)

$case\ \lambda x.cos(F(x,x)) * diff(\lambda y.F(x,y),x)\ of\ cos : ok \to^? X_2,$
$\qquad\qquad case\ X_2\ of\ ok : ok \to^? X_1$

$\Rightarrow_{Case\ Eval}$ (Apply the rule for $*$ after lifting it with x)

$\lambda x.case\ diff(\lambda y.F(x,y),x)\ of\ 1 : cos(F(x,x)), \ldots \to^? X_3,$
$\qquad\qquad case\ X_3\ of\ cos : ok \to^? X_2, \ldots$

$\Rightarrow_{Case\ Eval}$ (Apply the rule for *diff* after lifting it with x)

$\lambda x.case\ \lambda y.F(x,y)\ of\ \lambda y.y : 1, \ldots \to^? X_4,$
$\qquad\qquad \lambda x.case\ X_4(x)\ of\ 1 : cos(F(x,x)), \ldots \to^? X_3, \ldots$

$\Rightarrow^{\{F \mapsto \lambda x,y.y\}}_{Projection}$ (**Instantiate** F by the projection to the second argument)

$\lambda x.case\ \lambda y.y\ of\ \lambda y.y : 1, \ldots \to^? X_4,\ \ \lambda x.case\ X_4(x)\ of\ 1 : cos(x), \ldots \to^? X_3, \ldots$

$\Rightarrow_{Case\ Select}$ (Only case selection is possible)

$\lambda x.1 \to^? X_4,\ \ \lambda x.case\ X_4(x)\ of\ 1 : cos(x), \ldots \to^? X_3, \ldots$

$\Rightarrow_{Bind}$ (Case expression is evaluated, bind result to X_4)

$\lambda x.case\ 1\ of\ 1 : cos(x), \ldots \rightarrow^? X_3,\ \ case\ X_3\ of\ cos : ok \rightarrow^? X_2, \ldots$

$\Rightarrow_{Case\ Select}$ (Only case selection is possible)

$\lambda x.cos(x) \rightarrow^? X_3,\ \ case\ X_3\ of\ cos : ok \rightarrow^? X_2,\ \ case\ X_2\ of\ ok : ok \rightarrow^? X_1$

$\Rightarrow_{Bind}$ (Case expression is evaluated, bind result to X_3)

$case\ cos\ of\ cos : ok \rightarrow^? X_2,\ \ case\ X_2\ of\ ok : ok \rightarrow^? X_1$

$\Rightarrow_{Case\ Select}$ (Only case selection is possible)

$ok \rightarrow^? X_2,\ \ case\ X_2\ of\ ok : ok \rightarrow^? X_1$

$\Rightarrow_{Bind}$ (Case expression is evaluated, bind result to X_2)

$case\ ok\ of\ ok : ok \rightarrow^? X_1 \quad \Rightarrow_{Case\ Select} \quad ok \rightarrow^? X_1 \quad \Rightarrow_{Bind} \quad \{\}$

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

As we mentioned above, the lifting in the Case Eval inference rule is necessary to handle the application of rewrite rules and free variables in a context with bound variables correctly. For instance, consider the rewrite step

$$\lambda x.f(x) \longrightarrow^{f(Y)\rightarrow g(Y)}_{\{Y\mapsto x\}} \lambda x.g(x)\ .$$

In this rewrite step, we bind the variable Y to a bound variable, which can be treated as a constant here. This is possible since only matching is performed and all free variables in the rewrite rule disappear after the rewrite step.

Our calculus applies rewrite steps by the Case Eval rule. For this example, we first apply a lifter $\sigma = \{Y \mapsto Y'(x)\}$ to the above rule. Then the substitution $Y' \mapsto \lambda x.x$ is used for modeling the above rewrite step in our narrowing calculus. Note that free variables which appear directly below the outermost function symbol in rewrite rules immediately disappear when the Case Eval rule is applied. In order to show the need for lifting, we need a somewhat more involved example.

Example 5.2
Consider the function f defined by the rewrite rule

$$f(\lambda y.c(G(y))) \rightarrow G$$

The corresponding *hdt* is

$$f(X) \rightarrow case\ X\ of\ \lambda y.c(G(y))) : G$$

Clearly, the goal $\lambda x.f(F)) \rightarrow^? \lambda x, y.y$ has the solution $\{F \mapsto \lambda y.c(y)\}$ since $\lambda x.f(\lambda y.c(y))$ reduces in one step to $\lambda x.(\lambda y.y)$.

To solve this goal in our calculus LNT, we add the rule $g(\lambda x, y.y) \rightarrow ok$ which has the *hdt*

$$g(Y) \rightarrow case\ Y\ of\ \lambda x, y.y : ok$$

and solve the initial goal $case\ g(\lambda x.f(F))\ of\ ok : ok \rightarrow^? Z_1$:

$case\ g(\lambda x.f(F))\ of\ ok : ok \rightarrow^? Z_1$

$\Rightarrow_{Case\ Eval}$ $case\ \lambda x.f(F)\ of\ \lambda x, y.y : ok \rightarrow^? Z_2,\ case\ Z_2\ of\ ok : ok \rightarrow^? Z_1$

$$\Rightarrow_{Case\ Eval} \quad \lambda x.case\ F\ of\ \lambda y.c(G'(x,y)) : G'(x) \to^? Z_3,\ case\ Z_3\ of\ \lambda x,y.y : ok \to^? Z_2,\ case\ Z_2\ of\ ok : ok \to^? Z_1$$

$$\Rightarrow^{\{F \mapsto \lambda y.c(H(y))\}}_{Imitation} \quad \lambda x.case\ \lambda y.c(H(y))\ of\ \lambda y.c(G'(x,y)) : G'(x) \to^? Z_3,\ \ldots$$

$$\Rightarrow_{Case\ Select} \quad \lambda x,y.H(y) \to^? Z_3,\ case\ Z_3\ of\ \lambda x,y.y : ok \to^? Z_2,\ \ldots$$

$$\Rightarrow_{Bind} \quad case\ \lambda x,y.H(y)\ of\ \lambda x,y.y : ok \to^? Z_2,\ case\ Z_2\ of\ ok : ok \to^? Z_1$$

$$\Rightarrow^{\{H \mapsto \lambda x.x\}}_{Projection} \quad case\ \lambda x,y.y\ of\ \lambda x,y.y : ok \to^? Z_2,\ case\ Z_2\ of\ ok : ok \to^? Z_1$$

$$\Rightarrow_{Case\ Select} \quad ok \to^? Z_2,\ case\ Z_2\ of\ ok : ok \to^? Z_1$$

$$\Rightarrow_{Bind} \quad \Rightarrow_{Case\ Select} \quad \Rightarrow_{Bind} \quad \{\}$$

Thus, this derivation computes the solution $\{F \mapsto \lambda y.c(y)\}$ for the free variable F. Note that in the second step the *hdt* for f is applied after lifting it with x. This lifting replaces the free variable G by $G'(x)$ which is necessary to ensure the shallowness of the case pattern and the correct application of Case Select in subsequent steps. Moreover, an expression like $G'(x,y)$ can be refined to one of the bound variables x or y by instantiating G' to the appropriate projection function (actually, G' is subsequently instantiated to project to its second argument).

The next proposition states that our higher-order lazy narrowing calculus LNT is indeed a conservative extension of (first-order) needed narrowing.

Proposition 5.3
If all functions and goals occurring in a derivation are first-order, LNT computations correspond to needed narrowing derivations and vice versa.

Proof
If all functions and goals are first-order, λ-abstractions and functional variables do not occur. Thus, the Function Guess rule is not applicable and the λ-binders in all rules of the calculus LNT in Figure 3 can be omitted (which makes the application of the Projection rule impossible). Thus, the rules Bind, Case Select and Case Eval become identical to the corresponding first-order rules in Figure 2. Moreover, after an application of the rule Imitation, only Case Select is applicable. Thus, Imitation plus Case Select is equivalent to Case Guess in the first-order case. Therefore, first-order derivations with the rules in Figure 3 are equivalent to derivations with the rules in Figure 2. This implies the proposition by Theorems 3.3 and 3.5. □

6 Correctness and Completeness

In this section we show soundness and completeness of our higher-order narrowing calculus LNT. Similarly to the completeness proof of first-order needed narrowing (Antoy *et al.*, 1994), we show completeness w.r.t. reductions where redexes are computed via definitional trees (called "LNT reductions"). As in the first-order case, we show that LNT reduction steps are needed for reduction to a constructor normal form. In the first-order case, it is known that iterated reduction of needed redexes computes the normal form, if it exists (Huet & Lévy, 1991). A similar result is strongly conjectured but not yet known for the higher-order case. Since LNT

reductions compute ground constructor terms for terminating inductively sequential HRS, the completeness result w.r.t. LNT reductions is sufficient for practical programs (note that the definition of infinite data structures does not require non-terminating rules in functional logic languages, because infinite data structures can also be defined by terminating rules and the use of existential variables, see (Prehofer, 1997), Section 8.1.5).

We first define LNT reductions and then lift LNT reductions to narrowing. In the following, we assume an inductively sequential HRS $\mathcal{R}$ and a dedicated 0-ary constructor c.

6.1 LNT Reductions

We define LNT reductions as a restriction of our higher-order lazy narrowing calculus LNT. Then we show that they are in fact needed. For modeling rewriting, the Guess rules (i.e., Imitation, Function Guess, Projection) are not needed since each of these rules computes a non-identity substitution: $S \overset{*}{\Rightarrow}^{\{\}}_{LNT} S'$ 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 $\mathcal{T}' = f(\overline{X_n}) \to \mathcal{X}$, 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 **labelled tree** for a rule $\mathcal{T} = f(\overline{X_n}) \to \mathcal{X}$:

- $l_{\mathcal{T}}(p_f : case\ X\ of\ \overline{\mathcal{T}_n}) = p_f : case_p\ X\ of\ \overline{l_{\mathcal{T}}(\mathcal{T}_n)}$
  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 sequel that definitional trees for some inductively sequential HRS $\mathcal{R}$ are labelled. For instance, a labelled tree for the function $\leq$ of Example 1.1 is

$$\begin{array}{lllllll} X \leq Y \ \to & case_1\ X\ of & 0 & : & true, & & \\ & & s(X_1) & : & case_2\ Y\ of & 0 & : \ true, \\ & & & & & s(Y_1) & : \ X_1 \leq Y_1 \end{array}$$

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

Theorem 6.1
For an initial goal with $case_\epsilon\ t\ of\ c : c \to^?\ X_1 \overset{*}{\Rightarrow}^{\{\}}_{LNT} S$, S is of one of the following two forms:

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

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

Proof
Simple by induction on the LNT reduction. □

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 sometimes omitted in goal systems if no ambiguities may arise.

Assumption. We assume in the following that all goal systems are generated by LNT from some initial goal and are hence of one of the two forms of Theorem 6.1.

The next goal is to relate LNT derivations and rewriting.

Definition 6.2
We define an **associated substitution** for each goal system inductively on $\stackrel{*}{\Rightarrow}_{LNT}$:

- For an initial goal system of the form $S = case_\epsilon\ t\ of\ c : c \to^? X$, 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.

To see the need for this, consider for instance the rewrite system $g \to a$. For solving the goal $c(c(g)) \to^? c(c(a))$, we first have to add the rule $f(c(c(a))) \to ok$. In order to find the first needed position in the initial goal, we have the following computation:

$case_\epsilon\ f(c(c(g)))\ of\ ok\ :\ ok \to^? X_1$

$\Rightarrow_{Case\ Eval}$

$case_1\ c(c(g))\ of\ c(X_3) :\ case_{1.1}\ X_3\ of\ c(X_4)\ :\ case_{1.1.1}\ X_4\ of\ a\ : ok \to^? X_2,$
$\quad case_\epsilon\ X_2\ of\ ok\ :\ ok \to^? X_1$

$\Rightarrow_{Case\ Select}$

$case_{1.1}\ c(g)\ of\ c(X_4)\ :\ case_{1.1.1}\ X_4\ of\ a\ : ok \to^? X_2,$
$\quad case_\epsilon\ X_2\ of\ ok\ :\ ok \to^? X_1$

$\Rightarrow_{Case\ Select}$

$case_{1.1.1}\ g\ of\ a\ : ok \to^? X_2,\ case_\epsilon\ X_2\ of\ ok\ :\ ok \to^? X_1$

$\Rightarrow_{Case\ Eval}$

$a \to^? X_5,\ case_{1.1.1}\ X_5\ of\ a\ : ok \to^? X_2,\ case_\epsilon\ X_2\ of\ ok\ :\ ok \to^? X_1$

The associated substitution is $\{X_1 \mapsto f(c(c(g))), X_2 \mapsto f(c(c(g))), X_5 \mapsto g\}$. This can now be used to reconstruct the position of the redex g in the original term. Furthermore, we can translate the above goal system produced by LNT into one term. The idea is that $case_p\ t\ of\ \ldots \to^? X$ should be interpreted as the replacement of the case argument term t at position p in $\theta_S X$, i.e., $(\theta_S X)[t]_p$. Thus we can reconstruct the initial term $f(c(c(g)))$ by the position label and the associated substitution for X_5, X_2 and X_1. We cannot simply use the solution for X_1, since this reconstruction has to follow the rewrite steps. For instance, in the next step the Bind rule is applied to the last goal above. This corresponds to the execution of the rewrite step $f(c(c(g))) \longrightarrow f(c(c(a)))$. Hence we need to reconstruct the term $f(c(c(a)))$ from the resulting goal system which is of the following form:

$$case_{1.1.1}\ a\ of\ a\ : ok \to^? X_2, case_\epsilon\ X_2\ of\ ok\ :\ ok \to^? X_1 \qquad (1)$$

Now we can apply the same reconstruction and obtain $f(c(c(a)))$. In this way, we can reconstruct a rewrite step by a sequence of LNT steps. Formalizing this reconstruction for goal systems is the goal of the following technical treatment. We first define the translation of goal systems into terms.

Definition 6.3
For a goal system S of the form

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

(where $[r \to^? X,]$ is optional) with associated substitution θ, we define the **associated term** $\mathcal{A}(S)$ 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 $S_1 = case_\epsilon\ t\ of\ c : c \to^? X$, then $\mathcal{A}(S_1) = t$. The term associated to the goal (1) above is

$$f(c(c(g)))[\,f(c(c(g)))[a]_{1.1.1}\,]_\epsilon = f(c(c(a)))\ .$$

For a goal system S, we write $S{\downarrow}$ for the normal form obtained by applying Case Eval and Case Select. Observe via the last example that the steps in $S{\downarrow}$ correspond to finding the first, needed redex. We define an associated substitution for the intermediate variables $\overline{X_n}$ of a system of goals produced by LNT.

Lemma 6.4
$S{\downarrow}$ is well defined, i.e., computing $S{\downarrow}$ terminates and yields a unique goal system.

Proof
Termination follows easily since the size of the term used for rule selection in the leftmost case construct decreases. Note that the substitutions used for Case Select only bind variables to subterms of the leftmost term in the leftmost case construct. As Case Select and Case Eval do not apply simultaneously, uniqueness follows. □

Lemma 6.5

For a goal system S, the rules Case Eval and Case Select do not change the associated term.

Proof
We first establish the following invariant: If

$$\mathcal{I}_c(t) \stackrel{*}{\Rightarrow}^{\theta}_{LNT} case_{p_n}\ s\ of\ \ldots \to^? X_n, G\ =: S,$$

then $\theta\theta_S X_n|_{p_n} = s$ holds. This invariant is shown by induction on $\stackrel{*}{\Rightarrow}_{LNT}$. It is trivial for the Guess rules and follows from the definition of θ_S for Case Eval and Bind. Only the Case Select rule is more involved. Case Select reduces the term in the leftmost case construct:

$$S = case_p\ v(\overline{t_n})\ of\ \ldots \to^? X, \ldots \Rightarrow case_{p.p'}\ t_i\ of\ \ldots \to^? X, \ldots\ =: S'$$

Since $\theta_S X|_p = v(\overline{t_n})$ and $v(\overline{t_n})|_{p'} = t_i$, $\theta_{S'} X|_{p.p'} = t_i$ holds.

With the above invariant the theorem follows from the definition of $\mathcal{A}(S)$, since only the leftmost case construct is changed by each rule. □

From this result, we can infer $\mathcal{A}(S) = \mathcal{A}(S\downarrow)$ and $\mathcal{A}(\mathcal{I}_c(t)) = \mathcal{A}(\mathcal{I}_c(t)\downarrow)$.

Corollary 6.6
For a term t, we have $t = \mathcal{A}(\mathcal{I}_c(t)\downarrow)$.

Stability of reduction under substitution can be shown since LNT computations are outermost.

Lemma 6.7
For a term t, if $\mathcal{I}_c(t) \stackrel{*}{\Rightarrow}^{\{\}}_{LNT} \{\}$, then $\mathcal{I}_c(\theta t) \stackrel{*}{\Rightarrow}^{\{\}}_{LNT} \{\}$.

Proof
In $\mathcal{I}_c(t) \stackrel{*}{\Rightarrow}^{\{\}}_{LNT} \{\}$, no variable in $\mathcal{D}om(\theta)$ can affect the LNT computation (e.g., reduction cannot take place below a free variable). Hence $\mathcal{I}_c(\theta t) \stackrel{*}{\Rightarrow}^{\{\}}_{LNT} \{\}$ follows easily by induction on the length of the reduction. □

The next goal is to relate a rewrite step with LNT computations. In this vein, we show that for a given (part of a) rewrite rule, LNT finds the right branch in the corresponding definitional tree. For this we first give an auxiliary lemma which establishes this result for any term which corresponds to a path in a definitional tree, as defined via the function $pat_{\mathcal{T}}$.

Lemma 6.8
Assume $pat_{\mathcal{T}}(p_i : \mathcal{X}_i) = l$ for a definitional tree $\mathcal{T}$. There exists a reduction

$$\begin{array}{l} \lambda\overline{x}.case\ \lambda\overline{y}.l\ of\ \overline{\mathcal{T}_m} \to^? X, G \\ \quad \Rightarrow_{Case\ Eval}\ \stackrel{*}{\Rightarrow}_{Case\ Select}\ \lambda\overline{x}.case\ \lambda\overline{y}.p_i\ of\ \overline{p_n : \mathcal{X}_n} \to^? X', \\ \qquad\qquad\qquad\qquad \lambda\overline{x}.case\ \lambda\overline{y}.X'(\overline{x},\overline{y})\ of\ \overline{\mathcal{T}_m} \to^? X, G \\ \qquad\quad \Rightarrow_{Case\ Select} \qquad \lambda\overline{x}.\sigma(\mathcal{X}_i) \to^? X', \lambda\overline{x}.case\ \lambda\overline{y}.X'(\overline{x},\overline{y})\ of\ \overline{\mathcal{T}_m} \to^? X, G \end{array}$$

for some substitution σ.

Proof

First note that Case Eval applies if $l = f(\overline{t})$ for some defined symbol f with definitional tree $\mathcal{T}$. The claim follows easily by induction on the Case Select applications, by composing the substitutions computed for the variables in the selectors of $\mathcal{T}$. □

Now we can show easily that LNT detects a rewrite step when invoked with an instance of a left-hand side. Recall that the Bind rule applies to the resulting goal system of the LNT-computation in the next result. As shown in the above examples, the Bind rule, roughly speaking, executes the replacement step of a rewrite rules.

Corollary 6.9
There exists a rule $f(\overline{t}) \to r$ with $l = \sigma f(\overline{t})$ for some non-case term r iff

$$\lambda\overline{x}.case\ \lambda\overline{y}.l\ of\ \overline{\mathcal{T}_n} \to^? X, G$$
$$\Rightarrow_{Case\ Eval} \overset{*}{\Rightarrow}_{Case\ Select} \lambda\overline{x},\overline{y}.\sigma r \to^? X', \lambda\overline{x}.case\ \lambda\overline{y}.X'(\overline{x},\overline{y})\ of\ \overline{\mathcal{T}_n} \to^? X, G$$

Proof
The proof follows easily from Lemma 6.7 and Lemma 6.8. (Recall that the rules do not overlap and hence for a position in a term, only one rule applies.) □

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

Lemma 6.10
Let $S = \mathcal{I}_c(t)$. If $S{\downarrow}$ is of the form of Invariant 2 of Theorem 6.1, then $t = \mathcal{A}(S{\downarrow})$ is reducible at position $p = p_1 \cdots p_n$, where p_i are determined by the invariant of Theorem 6.1. Furthermore, if $t \longrightarrow_p t'$, then $\mathcal{I}_c(t'){\downarrow} = Bind(S{\downarrow}){\downarrow}$.

Proof
The only way S is transformed into $S{\downarrow}$ of the form of Invariant 2 is when a leftmost case construct, created by Case Eval (or the initial construct), is fully reduced to a term by Case Select without any intervening Case Eval applications. Say S is first transformed into

$$case_{p_n}\ s\ of\ \ldots \to^? X_n, \ldots, case_{p_1}\ X_2\ of\ c : c \to^? X_1\ =: S'$$

with $p = p_1 \cdots p_n$ and $t|_p = s$ such that

$$S' \Rightarrow_{Case\ Eval} \overset{*}{\Rightarrow}_{Case\ Select} r \to^? X_{n+1}, case_{p_n}\ X_{n+1}\ of\ \ldots \to^? X_n, \ldots\ =: S''$$

Since each path of a definitional tree corresponds to a left-hand side, $t|_p$ is reducible by Corollary 6.9. To show $\mathcal{I}_c(t'){\downarrow} = Bind(S{\downarrow}){\downarrow}$, consider the computation for $\mathcal{I}_c(t'){\downarrow}$. Since t and t' differ only at position p, i.e., $t' = t[\sigma r']_p$ for some rule $l' \to r'$, $\mathcal{I}_c(t')$ is transformed by LNT to

$$case_{p_n}\ \sigma r'\ of\ \ldots \to^? X_n, \ldots, case_{p_1}\ X_2\ of\ c : c \to^? X_1\ =: S_1.$$

Since $\mathcal{A}(S'') = t$, $\mathcal{A}(Bind(S'')) = t'$ follows from Lemma 6.5 and from the definition of the Bind rule. Hence $r = \sigma r'$ and $S'' \Rightarrow_{Bind} S_1$ follow. By uniqueness of normal forms, $Bind(S{\downarrow}){\downarrow} = Bind(S''){\downarrow} = S_1{\downarrow} = \mathcal{I}_c(t'){\downarrow}$ follows. □

Now, we can define LNT reductions:

Definition 6.11
A term t has a LNT redex at position p if $\mathcal{I}_c(t)\downarrow$ is of Invariant 2 of Theorem 6.1 with $p = p_1 \cdots p_n$ as in Theorem 6.1. We denote by $t \longrightarrow_{LNT} t'$ a **LNT reduction** step, i.e., a rewrite step $t \longrightarrow_p t'$ where p is the position of a LNT redex.

Thus, a LNT reduction step is a rewrite step at a position determined by the definitional trees (which are computed via our calculus LNT). For instance, consider Example 1.1 together with the rule $f(0) \to 0$ and the term $t = f(0) \leq f(0)$. Then $t \longrightarrow_1 0 \leq f(0)$ is a LNT reduction step in contrast to $t \longrightarrow_2 f(0) \leq 0$.

Theorem 6.12
For a term t, $\mathcal{I}_c(t) \stackrel{*}{\Rightarrow}^{\{\}}_{LNT} \{\}$ iff $t \stackrel{*}{\longrightarrow}_{LNT} c$.

Proof
If $t \stackrel{*}{\longrightarrow}_{LNT} c$, we can show $\mathcal{I}_c(t) \stackrel{*}{\Rightarrow}^{\{\}}_{LNT} \{\}$ easily by induction on the length of $t \stackrel{*}{\longrightarrow}_{LNT} c$ via Lemma 6.10.

Assuming $\mathcal{I}_c(t) \stackrel{*}{\Rightarrow}^{\{\}}_{LNT} \{\}$, we can show as in Lemma 6.10 that the reduction starts with computing $\mathcal{I}_c(t)\downarrow$ and then the Bind rule must apply. Similar to Lemma 6.10, we can show $t \longrightarrow_{LNT} t'$. By induction this yields a reduction $t \stackrel{*}{\longrightarrow}_{LNT} c$, since $\mathcal{I}_c(t) \stackrel{*}{\Rightarrow}_{LNT} case\ c\ of\ c : c \to^? X$ is the only way to transform $\mathcal{I}_c(t)$ to $\{\}$. □

It remains to show that LNT reductions are needed to compute a constructor-headed term. For a normalization result for the first-order case, we refer to (Middeldorp, 1997).

Theorem 6.13
If t reduces to c, then t has a LNT redex at a position p and t must be reduced at p eventually. Otherwise, t is not reducible to c.

Proof
In the computation of $\mathcal{I}_c(t)\downarrow$, we have goal systems of the form

$$case_{p_n}\ s\ of\ \ldots \to^? X_n, \ldots, case_{p_1}\ X_2\ of\ c : c \to^? X_1$$

where the term in the leftmost case construct, here s, is a subterm of t. We show that in this traversal of t, all subterms in the leftmost goals must be reduced at root position for t to reduce to c, or are constructor/bound variable headed subterms of such a redex. We state the proof in terms of this traversal of subterms of t; then we argue that $\mathcal{I}_c(t)\downarrow$ is of Invariant 2 of Theorem 6.1 and hence t has a redex by Lemma 6.10.

Starting at the root of t, $t = c$ or its root is a defined symbol. Otherwise, it is obviously not reducible to c. Since R is inductively sequential, there is a single rewrite rule $l \to r$ such that $t \stackrel{*}{\longrightarrow}_{\neq\epsilon} t' \longrightarrow^{l \to r}_{\epsilon} t''$. The traversal of t either ends in Invariant 2 of Theorem 6.1, if there is a redex at the root, or reaches another defined symbol where Case Eval applies. In the latter case, we apply the same argument as above for the root, only generalized to a position p where no rewrite rule applies

on a position on the path to p. Similarly, we show that there is a single applicable rule or t is not reducible to c.

By induction we either terminate with a redex which must be reduced for t to reduce to c, or we have a path which cannot be contracted and hence t is not reducible to c. □

The next desirable result is to show that LNT reductions are normalizing. This is suggested from related works (Oostrom, 1994; Klop, 1980), but is beyond the scope of this paper. However, the previous theorem implies that LNT reductions always compute the value in case of terminating rewrite systems.

Corollary 6.14
If $\mathcal{R}$ is terminating and t reduces to c, then $t \xrightarrow{*}_{LNT} c$.

6.2 Lifting Rewriting to Narrowing

We first take a closer look at the variables involved for a LNT computation.

Lemma 6.15
If $\mathcal{I}_c(t) \overset{*}{\Rightarrow}^{\theta}_{LNT} S \Rightarrow^{\theta'}_{LNT} S'$, then $\mathcal{D}om(\theta') \subseteq \mathcal{FV}(\theta t)$.

Proof
The substitution θ is composed of (partial) substitutions σ computed via one of the Guess rules. Since each such σ maps a variable occurring in the associated term $\mathcal{A}(S')$, the claim follows easily by induction on $\overset{*}{\Rightarrow}^{\theta}_{LNT}$. □

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

Lemma 6.16
If $S \overset{*}{\Rightarrow}^{\theta}_{LNT} \{\}$, then $\theta S \overset{*}{\Rightarrow}^{\{\}}_{LNT} \{\}$.

Proof
by induction on the length of $\overset{*}{\Rightarrow}_{LNT}$. Assume $S \Rightarrow^{\theta'} S' \overset{*}{\Rightarrow}^{\theta''} \{\}$. By induction hypothesis $\theta'' S' \overset{*}{\Rightarrow}^{\{\}}_{LNT} \{\}$. First, we show $\theta' S \overset{*}{\Rightarrow} S'$ by the following case distinction: If one of the Guess rules was used in $S \Rightarrow^{\theta'} S'$, then $\theta' S = S'$. Otherwise, $\theta' = \{\}$. Hence we have $\theta' S \overset{*}{\Rightarrow}^{\{\}} S'$, $\theta'' S' \overset{*}{\Rightarrow}^{\{\}} \{\}$, and infer $\theta''\theta' S \overset{*}{\Rightarrow}^{\{\}} \theta'' S' \overset{*}{\Rightarrow}^{\{\}} \{\}$ from Lemma 6.7. □

Theorem 6.17 (Correctness of LNT narrowing)
If $\mathcal{I}_c(t) \overset{*}{\Rightarrow}^{\theta}_{LNT} \{\}$ for a term t, then $\theta t \xrightarrow{*} c$.

Proof
First, $\mathcal{I}_c(\theta t) \overset{*}{\Rightarrow}_{LNT} \{\}$ follows from Lemma 6.16 and $\theta t \xrightarrow{*}_{LNT} c$ from Theorem 6.12. Since LNT reduction steps are particular rewrite steps, we conclude $\theta t \xrightarrow{*} c$. □

We first state completeness w.r.t. LNT computations.

Lemma 6.18

If $\theta S \stackrel{*}{\Rightarrow}^{\{\}}_{LNT} \{\}$ where θ is in $\mathcal{R}$-normal form and contains no dummies of S, i.e., $\mathcal{FV}(\theta) \cap \mathcal{FV}(S) = \mathcal{FV}(\mathcal{A}(S))$, then $S \stackrel{*}{\Rightarrow}^{\theta'}_{LNT} \{\}$ with $\theta' \leq_{\mathcal{FV}(\mathcal{A}(S))} \theta$.

Proof
From the given derivation we construct a reduction $S \stackrel{*}{\Rightarrow}^{\theta'}_{LNT} \{\}$, which is possibly longer, since Guess rules are interspersed. For this process to terminate, we need the following termination ordering. The ordering consists of the lexicographic combination of (A) the length of the LNT reduction $\theta S \stackrel{*}{\Rightarrow}^{\{\}}_{LNT} \{\}$ and (B) the sizes of the multiset of terms in the solutions for the variables in $\mathcal{D}om(\theta)$. (See (Baader & Nipkow, 1998; Dershowitz & Jouannaud, 1990) for a definition of lexicographic orderings.) We have the following cases depending on the form of S:

- If $S = e \rightarrow^? X, G$ and on θS the Bind rule applies, then Bind applies to S as well since $X \notin \mathcal{D}om(\theta)$. Since $\theta Bind(S) = Bind(\theta S)$, the induction hypothesis applies with a shorter reduction, decreasing A.
- If $S = case\ \lambda\overline{x}.v(\overline{t})\ of \ldots, G$, then either Case Select or Case Eval must apply on θS and hence on S as well. The induction hypothesis applies as in the last case.
- If $S = \lambda\overline{x}.case\ \lambda\overline{y}.X(\overline{t})\ of \ldots, G$, then $\theta X(\overline{t})$ must be of one of the following forms:
  - $\lambda\overline{x}.c(\overline{t'})$ such that Case Select applies on θS. In this case, Imitation is applicable with a binding σ such that $\exists\theta'.\theta = \theta'\sigma$ as in proof of higher-order unification (see (Snyder & Gallier, 1989; Prehofer, 1997)). Since θ' is $\mathcal{R}$-normalized and dummy free, the induction hypothesis applies with a smaller solution, decreasing B.
  - $\lambda\overline{x}.x(\overline{t'})$ such that Case Select applies on θS. This case proceeds as the above with Projection instead of Imitation.
  - $\lambda\overline{x}.f(\overline{t'})$ such that Case Eval applies on θS. Here, Function Guess applies. The case concludes as the two above. For the precondition of the rule we observe that if $\lambda\overline{x},\overline{y}.X(\overline{t})$ is a higher-order pattern, then $\theta\lambda\overline{x},\overline{y}.X(\overline{t})$ is not $\mathcal{R}$-reducible (as shown in (Prehofer, 1997)) and hence $\theta S \stackrel{*}{\Rightarrow}^{\{\}}_{LNT} \{\}$ is impossible.

□

Theorem 6.19 (*Relative completeness of LNT narrowing*)
If $\theta t \stackrel{*}{\longrightarrow}_{LNT} c$ and θ is in $\mathcal{R}$-normal form, then $\mathcal{I}_c(t) \stackrel{*}{\Rightarrow}^{\theta'}_{LNT} \{\}$ with $\theta' \leq_{\mathcal{FV}(t)} \theta$.

Proof
Completeness follows from Theorem 6.12 and the previous lemma. □

Although the normalization property of LNT reductions is only known for the first-order case (by Proposition 5.3 and the results in (Antoy, 1992; Antoy *et al.*, 1994)), we can state a completeness result of LNT narrowing in the higher-order case for terminating rewrite systems.

Theorem 6.20 (*Completeness of LNT narrowing*)
If $\mathcal{R}$ is terminating, $\theta t \stackrel{*}{\longrightarrow} c$, and θ is in $\mathcal{R}$-normal form, then $\mathcal{I}_c(t) \stackrel{*}{\Rightarrow}^{\theta'}_{LNT} \{\}$ with $\theta' \leq_{\mathcal{FV}(t)} \theta$.

Proof
Follows from the previous theorem and Corollary 6.14. □

7 Optimality regarding Solutions

We show here another important aspect, namely uniqueness of the computed solutions. Compared to the more general case in (Prehofer, 1997), 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.

We call two substitutions σ and σ' **independent** on a set of variables V if there exists a variable $x \in V$ such that σx and $\sigma' x$ are not unifiable. Now we can show that different derivations of our calculus always compute independent solutions.

Theorem 7.1 (Optimality)
If $\mathcal{I}_c(t) \stackrel{*}{\Rightarrow}^{\theta}_{LNT} \{\}$ and $\mathcal{I}_c(t) \stackrel{*}{\Rightarrow}^{\theta'}_{LNT} \{\}$ are two different derivations, then θ and θ' are independent on $\mathcal{FV}(t)$.

Proof
The claim follows from examining the substitutions computed. First, it is to observe that, except for the Guess rules, no rule overlap, i.e., apply simultaneously to a particular goal system. Thus, the two derivations have the form

$$\mathcal{I}_c(t) \;\stackrel{*}{\Rightarrow}^{\theta_0}_{LNT}\; S \;\stackrel{*}{\Rightarrow}^{\theta_1}_{LNT}\; S_1 \;\stackrel{*}{\Rightarrow}^{\theta'_1}_{LNT}\; \{\}$$

and

$$\mathcal{I}_c(t) \;\stackrel{*}{\Rightarrow}^{\theta_0}_{LNT}\; S \;\stackrel{*}{\Rightarrow}^{\theta_2}_{LNT}\; S_2 \;\stackrel{*}{\Rightarrow}^{\theta'_2}_{LNT}\; \{\}$$

where the steps $S \stackrel{*}{\Rightarrow}^{\theta_1}_{LNT} S_1$ and $S \stackrel{*}{\Rightarrow}^{\theta_2}_{LNT} S_2$ are different applications of a Guess rule (otherwise, the derivations cannot be different). These different applications of a Guess rule compute independent bindings for a variable X, where the following bindings are possible:

$$\begin{array}{ll}
\{X \mapsto \lambda\overline{x_n}.c(\overline{H_m(\overline{x_n})})\} & \text{(Imitation)} \\
\{X \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\} & \text{(Function Guess)} \\
\{X \mapsto \lambda\overline{x_n}.x_i(\overline{H_m(\overline{x_n})})\} & \text{(Projection)}
\end{array}$$

By Lemma 6.15, these independent bindings are bindings for a variable $X \in \mathcal{FV}(\theta_0 t)$. Thus, there is a variable $Y \in \mathcal{FV}(t)$ such that $\{Y \mapsto \theta_1(\theta_0(Y)\}$ and $\{Y \mapsto \theta_2(\theta_0(Y)\}$ are independent on $\mathcal{FV}(t)$. As a consequence, $\theta = \theta'_1\theta_1\theta_0$ and $\theta = \theta'_2\theta_2\theta_0$ are independent on $\mathcal{FV}(t)$. □

It is also conjectured that our notion of needed reductions is optimal (this is subject to current research (Asperti & Laneve, 1994; Oostrom, 1994; Oostrom, 1996)). Note, however, that sharing is needed for optimality, as shown for the first-order case in (Antoy *et al.*, 1994).

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 (Avenhaus & Loría-Sáenz, 1994) is easy to see in our context, although lifting over binders obscures the results somewhat unnecessarily.[5] 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 these are immediately eliminated when *hdt*s are introduced in the Case Eval rule. For instance, the rule

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

is weakly higher-order, if X and R are first-order.

Theorem 8.1
If $\mathcal{I}_c(t) \stackrel{*}{\Rightarrow}_{LNT} S$ where t is quasi first-order w.r.t. weakly higher-order rules, then $\mathcal{A}(S)$ is quasi first-order.

Proof
We establish the claim by induction on $\stackrel{*}{\Rightarrow}$. Assume $S \Rightarrow S'$. First, we show that only higher-order patterns occur in S'. The only rule where non-patterns are involved is the Case Eval rule. In this rule, all $\overline{X_n}$ of a weakly higher-order rule $f(\overline{X_n}) \to \mathcal{X}$ are bound to quasi first-order terms by σ, hence all terms in $\sigma\mathcal{X}$ are higher-order patterns.

Furthermore, it is to show that $\mathcal{A}(S')$ is quasi first-order. Since the Guess rules are first-order in this case, only the Bind rule must be considered: As all variables in the right-hand side in the leaf must occur in the selectors on the path to the leaf, all its variables must have been bound before Bind applies. Since the variables in selectors are only bound to (sub-)terms of $\mathcal{A}(S')$ (in the Case Select rule), the right-hand side is instantiated to a quasi first-order term. □

As a consequence of the last result, Function Guess and Projection do not apply and Imitation is only used as in the first-order case. For instance, a term like $map(f, l)$ can be solved without higher-order unification (i.e., as in Section 3) provided that f is a defined function symbol and l is a list of first-order objects.

9 Conclusions

We have presented an effective model for the integration of higher-order functional and logic programming with completeness and optimality results. The completeness results are stated w.r.t. LNT reductions which we have defined via definitional

[5] Considering liftings is missing in (Loría-Sáenz, 1993)

trees. In the first-order case, LNT reductions are identical to needed reductions. In the higher-order case, our computation model is complete for terminating rewrite systems, complete w.r.t. LNT reductions in the presence of non-terminating functions, and optimal w.r.t. the number of computed solutions. Since we permit higher-order logical variables and λ-abstractions, our strategy is a suitable basis for truly higher-order functional logic languages, i.e., declarative languages that provide the synthesis of values for first-order as well as higher-order variables. 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 (Prehofer, 1997) to this refined context.

Acknowledgements. The authors are grateful to the anonymous referees for their detailed comments on a previous version of this paper which were very helpful to improve the readability of the paper.

References

Antoy, S. (1992). Definitional trees. *Pages 143–157 of: Proc. of the 3rd International Conference on Algebraic and Logic Programming.* Springer LNCS 632.

Antoy, S. (1996). Needed narrowing in Prolog. *Pages 473–474 of: Proc. of the 8th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96).* Springer LNCS 1140.

Antoy, S., Echahed, R., & Hanus, M. (1994). A needed narrowing strategy. *Pages 268–279 of: Proc. 21st ACM Symposium on Principles of Programming Languages.*

Asperti, A., & Laneve, C. (1994). Interaction systems I: The theory of optimal reductions. *Mathematical Structures in Computer Science*, **4**, 457–504.

Avenhaus, J., & Loría-Sáenz, C. A. (1994). Higher-order conditional rewriting and narrowing. Jouannaud, Jean-Pierre (ed), *1st International Conference on Constraints in Computational Logics.* München, Germany: Springer LNCS 845.

Baader, F., & Nipkow, T. (1998). *Term rewriting and all that.* Cambridge University Press.

Barendregt, H.P. (1984). *The lambda calculus, its syntax and semantics.* 2nd edn. North Holland.

Dershowitz, N., & Jouannaud, J.-P. (1990). Rewrite systems. *Pages 243–320 of:* Leeuwen, Jan Van (ed), *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics.* Elsevier.

Giovannetti, E., Levi, G., Moiso, C., & Palamidessi, C. (1991). Kernel LEAF: A logic plus functional language. *Journal of Computer and System Sciences*, **42**(2), 139–185.

González-Moreno, J.C., Hortalá-González, M.T., & Rodríguez-Artalejo, M. (1992). On the completeness of narrowing as the operational semantics of functional logic programming. *Pages 216–230 of: Proc. Computer Science Logic '92.* Springer LNCS 702.

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

Hanus, M. (1995). Efficient translation of lazy functional logic programs into Prolog. *Pages 252–266 of: Proc. Fifth International Workshop on Logic Program Synthesis and Transformation.* Springer LNCS 1048.

Hanus, M. (1997). A unified computation model for functional and logic programming. *Pages 80–93 of: Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris).*

Hindley, J.R., & Seldin, J. P. (1986). *Introduction to combinators and λ-calculus.* Cambridge University Press.

Hölldobler, S. (1989). *Foundations of equational logic programming.* Springer LNCS 353.

Hudak, P., Peyton Jones, S., & Wadler, P. (1992). Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, **27**(5). Version 1.2.

Huet, G., & Lévy, J.-J. (1991). Computations in orthogonal rewriting systems, I. *Pages 395–414 of:* Lassez, J.-L., & Plotkin, G. (eds), *Computational Logic: Essays in Honor of Alan Robinson.* Cambridge, MA: MIT Press.

Hullot, J.-M. (1980). Canonical forms and unification. *Pages 318–334 of: Proc. 5th Conference on Automated Deduction.* Springer LNCS 87.

Ida, T., & Nakahara, K. (1997). Leftmost outside-in narrowing calculi. *Journal of Functional Programming*, **7**(2), 129–161.

Klop, J.W. (1980). *Combinatory reduction systems.* Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam.

Lloyd, J.W. (1994). Combining functional and logic programming languages. *Proceedings of the 1994 International Logic Programming Symposium, ILPS'94.*

Loogen, R., Fraguas, F. Lopez, & Artalejo, M. Rodríguez. (1993). A demand driven computation strategy for lazy narrowing. *Pages 184–200 of: Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming.* Springer LNCS 714.

Loría-Sáenz, C. A. (1993). *A theoretical framework for reasoning about program construction based on extensions of rewrite systems.* Ph.D. thesis, Univ. Kaiserslautern.

Martelli, A., & Montanari, U. (1982). An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, **4**(2), 258–282.

Martelli, A., Rossi, G.F., & Moiso, C. (1989). Lazy unification algorithms for canonical rewrite systems. *Chap. 8, pages 245–274 of:* Aït-Kaci, Hassan, & Nivat, Maurice (eds), *Resolution of Equations in Algebraic Structures, Volume 2, Rewriting Techniques.* New York: Academic Press.

Middeldorp, A. (1997). Call by need computations to root-stable form. *Pages 94–105 of: Proc. 24th ACM Symposium on Principles of Programming Languages.*

Middeldorp, A., Okui, S., & Ida, T. (1996). Lazy narrowing: Strong completeness and eager variable elimination. *Theoretical Computer Science*, **167**(1,2), 95–130.

Miller, D. (1991). A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, **1**, 497–536.

Moreno-Navarro, J.J., & Rodríguez-Artalejo, M. (1992). Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, **12**, 191–223.

Nadathur, G., & Miller, D. (1988). An overview of λProlog. *Pages 810–827 of: Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle).* MIT Press.

Nakahara, K., Middeldorp, A., & Ida, T. (1995). A complete narrowing calculus for higher-order functional logic programming. *Pages 97–114 of: Proc. of the 7th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'95).* Springer LNCS 982.

Nipkow, T. (1991). Higher-order critical pairs. *Pages 342–349 of: Proc. 6th IEEE Symp. Logic in Computer Science.*

Oostrom, Vincent van. (1994). *Confluence for abstract and higher-order rewriting.* Ph.D. thesis, Vrije Universiteit. Amsterdam.

Oostrom, Vincent van. (1996). Higher-order families. Ganzinger, H. (ed), *Proc. Seventh International Conference on Rewriting Techniques and Applications (RTA'96)*. Springer LNCS 1103.

Peyton Jones, S.L. (1987). *The implementation of functional programming languages.* Prentice Hall.

Prehofer, C. (1994). Higher-order narrowing. *Pages 507–516 of: Proc. Ninth Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press.

Prehofer, C. (1995a). A Call-by-Need Strategy for Higher-Order Functional-Logic Programming. *Pages 147–161 of:* Lloyd, J. (ed), *Proc. of the 1995 International Symposium on Logic Programming.* MIT Press.

Prehofer, C. (1995b). Higher-order narrowing with convergent systems. *4th Int. Conf. Algebraic Methodology and Software Technology, AMAST '95.* Springer LNCS 936.

Prehofer, C. (1996). Some applications of functional logic programming. *Pages 35–45 of: Proc. JICSLP'96 Workshop on Multi-Paradigm Logic Programming.* TU Berlin, Technical Report No. 96-28.

Prehofer, C. (1997). *Solving higher-order equations: From logic to programming.* Birkhäuser PCTS Series.

Slagle, J.R. (1974). Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, **21**(4), 622–642.

Snyder, W., & Gallier, J. (1989). Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, **8**, 101–140.

A Appendix

This appendix contains the proofs omitted in Section 3.

A.1 Proof of Theorem 3.3

In order to prove the equivalence of needed narrowing and leftmost-outermost narrowing with case expressions, we have to relate both kinds of derivations. For this purpose, we define the following translation $\mathcal{EC}$ from $\mathcal{E}val$-goals into terms with case expressions.

$$\begin{array}{lcl}
\mathcal{EC}(t) & = & t \\
\mathcal{EC}(\mathcal{E}val(t,\mathcal{T})) & = & \sigma(\mathcal{C}ase(\mathcal{T})) \quad \text{where } \sigma(pat(\mathcal{T})) = t \\
\mathcal{EC}(G, \mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k}))) & = & case\ \mathcal{EC}(G)\ of\ \overline{p_k : \mathcal{X}_k} \\
 & & \text{where } \mathcal{EC}(\mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k}))) = case\ \ldots\ of\ \overline{p_k : \mathcal{X}_k}
\end{array}$$

Hence, a single $\mathcal{E}val$-goal is translated into the definitional tree represented by case expressions and instantiated with the arguments of the goal. A sequence of $\mathcal{E}val$-goals, which may occur due to nested function evaluations, is folded into a single case expression by inserting the first goals into the first argument of the final case expression. For instance, the $\mathcal{E}val$-goal

$$0,\ \mathcal{E}val(0+0 \leq Y, branch(X_1 \leq Y_1, 1, rule(0 \leq Y_1 \to true), branch(s(X_2) \leq Y_1, \ldots)))$$

is translated by $\mathcal{EC}$ into the term

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

The following lemma shows that each inference step in the needed narrowing calculus w.r.t. $\mathcal{R}$ corresponds to zero or one leftmost-outermost narrowing steps w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$.

Lemma A.1
Let G be an $\mathcal{E}val$-goal, $t = \mathcal{EC}(G)$, and $G \Rightarrow^{\sigma} G'$ be an inference step in the needed narrowing calculus. Then, either $\mathcal{EC}(G') = t$ and $\sigma = \{\}$, or there exists a unique leftmost-outermost narrowing step $t \leadsto_{\sigma} t'$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$ with $\mathcal{EC}(G') = t'$.

Proof
We distinguish the different cases for the applied inference rule of the needed narrowing calculus. Note that G has the form $\mathcal{E}val(s, \mathcal{T}), G_0$ except for the inference rules *Initial* and *Replace Subterm*.

1. The inference rule *Initial* is applied: Then $G = t = f(\overline{t_n})$ for some function f. Let $\mathcal{T}$ be the definitional tree for f with pattern $f(\overline{X_n})$ and $\varphi = \{\overline{X_n \mapsto t_n}\}$. Then $t \leadsto_{\{\}} \varphi(\mathcal{C}ase(\mathcal{T}))$ is a unique leftmost-outermost narrowing step w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$. Moreover, $\mathcal{EC}(G') = \mathcal{EC}(\mathcal{E}val(t, \mathcal{T})) = \varphi(\mathcal{C}ase(\mathcal{T}))$.
2. The inference rule *Apply* is used: Then $\sigma = \{\}$, $\mathcal{T} = rule(l \to r)$, $\varphi(l) = s$ for some substitution φ, and $G' = \varphi(r), G_0$. If $G_0 = \{\}$, then $t = \mathcal{EC}(G) = \varphi(r) = \mathcal{EC}(G')$ by definition of $\mathcal{EC}$. If $G_0 \neq \{\}$, then, by definition of $\mathcal{EC}$, t and $\mathcal{EC}(G')$ may only differ in the case argument of some case expression, where t contains the subterm $\mathcal{EC}(\mathcal{E}val(s, \mathcal{T}))$ and $\mathcal{EC}(G')$ contains the subterm $\varphi(r)$ at this case argument. However, $\mathcal{EC}(\mathcal{E}val(s, \mathcal{T})) = \varphi(r)$ by definition of $\mathcal{EC}$.
3. The inference rule *Select* is applied: Then $\mathcal{T} = branch(\pi, o, \overline{\mathcal{T}_k})$, $s|_o = c(\overline{t_n})$, $\sigma = \{\}$, and $G' = \mathcal{E}val(s, \mathcal{T}_i), G_0$ where $pat(\mathcal{T}_i) = c(\overline{X_n})$. First, consider the case $G_0 = \{\}$. Then

   $$t = \varphi(case\ \pi|_o\ of\ \overline{pat(\mathcal{T}_k)|_o : \mathcal{C}ase(\mathcal{T}_k)})$$

   with $\varphi(\pi) = s$. Since $s|_o = c(\overline{t_n})$, $\varphi(\pi)|_o = c(\overline{X_n})$. Due to the form of the case rules, exactly one case rule (the i-th rule) can be applied to t, i.e., $t \leadsto_{\{\}} \varphi'(\varphi(\mathcal{C}ase(\mathcal{T}_i)))$ with $\varphi' = \{\overline{X_n \mapsto t_n}\}$ and $pat(\mathcal{T}_i) = \pi[c(\overline{X_n})]_o$. Since $\varphi(\pi) = s$ and $s|_o = c(\overline{t_n})$, $\varphi'(\varphi(pat(\mathcal{T}_i))) = s$. Thus, $\mathcal{EC}(\mathcal{E}val(s, \mathcal{T}_i)) = \varphi'(\varphi(\mathcal{C}ase(\mathcal{T}_i)))$.
   If $G_0 \neq \{\}$, t is a term consisting of nested case expressions and $\mathcal{EC}(\mathcal{E}val(s, branch(\pi, o, \overline{\mathcal{T}_k})))$ is the leftmost-outermost position in t where a narrowing step can be applied (by definition of $\mathcal{EC}$). Hence we apply a leftmost-outermost narrowing step to this subterm of t analogously to the case $G_0 = \{\}$.
4. The inference rule *Instantiate* is applied: Then $\mathcal{T} = branch(\pi, o, \overline{\mathcal{T}_k})$, $s|_o = X$, $\sigma = \{X \mapsto pat(\mathcal{T}_i)\}$, and $G' = \sigma(\mathcal{E}val(s, \mathcal{T}_i), G_0)$. Consider the case $G_0 = \{\}$ (the case $G_0 \neq \{\}$ can be treated analogously as in the previous case). Then

   $$t = \varphi(case\ \pi|_o\ of\ \overline{pat(\mathcal{T}_k)|_o : \mathcal{C}ase(\mathcal{T}_k)})$$

   with $\varphi(\pi) = s$. Since $s|_o = X$ and due to the form of the case rules, exactly one case rule (the i-th rule) can be applied to t in order to instantiate X to the

same pattern, i.e., $t \leadsto_\sigma \sigma(\varphi(\mathcal{C}ase(\mathcal{T}_i)))$. Since $\varphi(\pi) = s$ and $\sigma(s|_o) = \sigma(X) = pat(\mathcal{T}_i)|_o$, $\sigma(\varphi(pat(\mathcal{T}_i))) = s$. Thus, $\mathcal{EC}(\mathcal{E}val(s, \mathcal{T}_i)) = \sigma(\varphi(\mathcal{C}ase(\mathcal{T}_i)))$.

5. The inference rule *Eval Subterm* is applied: Then $\mathcal{T} = branch(\pi, o, \overline{\mathcal{T}_k})$, $s|_o = f(\overline{t_n})$, $\sigma = \{\}$, and $G' = \mathcal{E}val(s|_o, \mathcal{T}'), \mathcal{E}val(s, \mathcal{T}), G_0$ where $\mathcal{T}'$ is the definitional tree of f. Consider the case $G_0 = \{\}$ (the case $G_0 \neq \{\}$ can be treated analogously). Then

$$t = \varphi(case\ \pi|_o\ of\ \overline{pat(\mathcal{T}_k)|_o : \mathcal{C}ase(\mathcal{T}_k)})$$

with $\varphi(\pi) = s$. Since $s|_o = f(\overline{t_n}) = \varphi(\pi)|_o$, no case rule is applicable to the root of t. Thus, $\varphi(\pi|_o)$ is the subterm at the leftmost-outermost narrowing position, and the only applicable rule is $f(\overline{X_n}) \to \mathcal{C}ase(\mathcal{T}')$ (if $f(\overline{X_n})$ is the pattern of $\mathcal{T}'$). Thus,

$$t \leadsto_{\{\}} case\ \tau(\mathcal{C}ase(\mathcal{T}'))\ of\ \overline{pat(\mathcal{T}_k)|_o : \varphi(\mathcal{C}ase(\mathcal{T}_k))}$$

with $\tau = \{\overline{X_n \mapsto t_n}\}$ is the only possible leftmost-outermost narrowing step. Moreover,

$$\begin{array}{ll} & \mathcal{EC}(\mathcal{E}val(s|_o, \mathcal{T}'), \mathcal{E}val(s, \mathcal{T})) \\ = & case\ \mathcal{EC}(\mathcal{E}val(s|_o, \mathcal{T}'))\ of\ \overline{pat(\mathcal{T}_k)|_o : \varphi(\mathcal{C}ase(\mathcal{T}_k))} \\ = & case\ \tau(\mathcal{C}ase(\mathcal{T}'))\ of\ \overline{pat(\mathcal{T}_k)|_o : \varphi(\mathcal{C}ase(\mathcal{T}_k))}\ . \end{array}$$

The last equality holds by definition of $\mathcal{EC}$ since $\tau(pat(\mathcal{T}')) = \tau(f(\overline{X_n})) = f(\overline{t_n}) = s|_o$.

6. The inference rule *Replace Subterm* is applied: Then $G = r, \mathcal{E}val(s, \mathcal{T}), G_0$, $\mathcal{T} = branch(\pi, o, \overline{\mathcal{T}_k})$, $\sigma = \{\}$, and $G' = \mathcal{E}val(s[r]_o, \mathcal{T}), G_0$. Consider the case $G_0 = \{\}$ (the case $G_0 \neq \{\}$ can be treated analogously). Then

$$\begin{array}{rcl} t & = & \mathcal{EC}(r, \mathcal{E}val(s, \mathcal{T})) \\ & = & case\ \mathcal{EC}(r)\ of\ \overline{pat(\mathcal{T}_k)|_o : \varphi(\mathcal{C}ase(\mathcal{T}_k))} \\ & = & case\ r\ of\ \overline{pat(\mathcal{T}_k)|_o : \varphi(\mathcal{C}ase(\mathcal{T}_k))} \end{array}$$

with $\varphi(\pi) = s$. On the other hand,

$$\begin{array}{ll} & \mathcal{EC}(\mathcal{E}val(s[r]_o, branch(\pi, o, \overline{\mathcal{T}_k}))) \\ = & case\ \varphi'(\pi|_o)\ of\ \overline{pat(\mathcal{T}_k)|_o : \varphi'(\mathcal{C}ase(\mathcal{T}_k))} \end{array}$$

with $\varphi'(\pi) = s[r]_o$. Hence the only difference between φ and φ' is the instantiation of the variable $\pi|_o$: $\varphi'(\pi|_o) = r$ and $\varphi(\pi|_o) = s|_o$. W.l.o.g. we can assume that the case variable $\pi|_o$ does not occur in any subtree $\overline{\mathcal{T}_k}$ (we can always construct the definitional tree in such a way). Thus, both terms t and $\mathcal{EC}(\mathcal{E}val(s[r]_o, branch(\pi, o, \overline{\mathcal{T}_k})))$ are identical (i.e., it is not necessary to perform a leftmost-outermost narrowing step).

□

The equivalence of needed narrowing w.r.t. $\mathcal{R}$ and leftmost-outermost narrowing w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$ is based on the previous lemma:

Theorem 3.3
Let t be a term and c be a 0-ary constructor. For each needed narrowing derivation

$t \leadsto^*_\sigma c$ w.r.t. $\mathcal{R}$ there exists a leftmost-outermost narrowing derivation $t \leadsto^*_\sigma c$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$, and vice versa.

Proof
By induction on the derivation steps and applying Lemma A.1, we can construct for each needed narrowing derivation starting from t a unique leftmost-outermost narrowing derivation starting from $\mathcal{EC}(t) = t$ which computes the same solution for the variables in t. On the other hand, it is straightforward to show (by a case distinction similar to the proof of Lemma A.1) that each leftmost-outermost narrowing derivation $t \leadsto^*_\sigma c$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$ corresponds to a needed narrowing derivation $t \leadsto^*_\sigma c$ w.r.t. $\mathcal{R}$. □

A.2 Proof of Theorem 3.5

To prove the equivalence of leftmost-outermost narrowing and the lazy narrowing calculus LNT, we need a few notions and technical lemmas to establish the precise equivalence between leftmost-outermost narrowing derivations and derivation in the lazy narrowing calculus LNT. First, note that the inference rules of the calculus LNT keep the following important invariant on goal systems:

(∗) If $G_1, l \to^? r, G_2$ is a goal system, then r is a variable not occurring in G_1 and l.

There is a strong correspondence between terms with case expressions and goal systems, since each single equation $t \to^? X$ can be "flattened" into a goal system by the following function $\mathcal{F}lat$:[6]

$$\mathcal{F}lat(f(\overline{t_n}) \to^? X) \;=\; f(\overline{t_n}) \to^? X$$

$$\mathcal{F}lat(case\ t\ of\ \overline{p_n : \mathcal{X}_n} \to^? X) = \begin{cases} \mathcal{F}lat(t \to^? Y), case\ Y\ of\ \overline{p_n : \mathcal{X}_n} \to^? X & \text{if } t = case \ldots \text{ and } Y \text{ fresh variable} \\ case\ t\ of\ \overline{p_n : \mathcal{X}_n} \to^? X & \text{otherwise} \end{cases}$$

For instance, if we apply the function $\mathcal{F}lat$ to the goal

$$case\ (case\ X \leq Y\ of\ true : true)\ of\ true : true \to^? T\ ,$$

we obtain the "flattened" goal system

$$case\ X \leq Y\ of\ true : true \to^? Z,\ case\ Z\ of\ true : true \to^? T\ .$$

On the other hand, goal systems satisfying invariant (∗) can be "folded" by the following function $\mathcal{F}old$ into a single equation representing a term with nested case expressions:

$$\begin{array}{lcll} \mathcal{F}old(t \to^? X) & = & t \to^? X & \\ \mathcal{F}old(t \to^? X, G) & = & \mathcal{F}old(\sigma(G)) & \text{with } \sigma = \{X \mapsto t\} \end{array}$$

[6] Formally, $\mathcal{F}lat$ is not a function due to the arbitrarily chosen fresh variables. However, by fixing the set of fresh variables and introducing an order on it, $\mathcal{F}lat$ can be interpreted as a function.

The following lemma shows that, for each leftmost-outermost narrowing step in a derivation w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$, there is a corresponding LNT-derivation w.r.t. $\mathcal{R}'$.

Lemma A.2
Let $t \leadsto_\sigma t'$ be a leftmost-outermost narrowing step in a derivation of the initial term $case\ t_0\ of\ c : c$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$ and X a fresh variable. Then there exists a LNT-step $\mathcal{F}lat(t \to^? X) \Rightarrow^\sigma G$ w.r.t. $\mathcal{R}'$ such that $\mathcal{F}old(G) = t' \to^? X$.

Proof
Since $t \leadsto_\sigma t'$ is a leftmost-outermost narrowing step in a derivation of the initial term $case\ t_0\ of\ c : c$, t has a *case* symbol at the top. Moreover, since it was derived by leftmost-outermost narrowing steps w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$, t has the structure

$$t = case_1\ (\ldots(case_{n-1}\ (case_n\ s\ of\ \overline{p_k : \mathcal{X}_k})\ of\ \ldots)\ldots)\ of\ \ldots$$

where s has not a *case* symbol at the top (here we use indices to distinguish the different case symbols). By definition of $\mathcal{F}lat$,

$$\mathcal{F}lat(t \to^? X)\ =\ case_n\ s\ of\ \overline{p_k : \mathcal{X}_k} \to^? Y_{n-1}, \ldots, case_1\ Y_1\ of\ \ldots \to^? X\ .$$

There are the following possibilities for s:

1. s is operation-rooted, i.e., $s = f(\overline{t_n})$. Then s is the subterm reduced by the leftmost-outermost narrowing step, $\sigma = \{\}$, and

   $$t' = case_1\ (\ldots(case_{n-1}\ (case_n\ \varphi(\mathcal{X})\ of\ \overline{p_k : \mathcal{X}_k})\ of\ \ldots)\ldots)\ of\ \ldots$$

   if $f(\overline{X_n}) \to \mathcal{X}$ is a rewrite rule and $\varphi = \{\overline{X_n \mapsto t_n}\}$. Since s is operation-rooted, we can only apply the *Case Eval* rule to the goal system $\mathcal{F}lat(t \to^? X)$:

   $\mathcal{F}lat(t \to^? X)\ \Rightarrow^{\{\}}$
   $\varphi(\mathcal{X}) \to^? Y_n, case_n\ Y_n\ of\ \overline{p_k : \mathcal{X}_k} \to^? Y_{n-1}, \ldots, case_1\ Y_1\ of\ \ldots \to^? X =: G$

   By definition of $\mathcal{F}old$, $\mathcal{F}old(G) = t' \to^? X$.
2. s is a variable: Then $case_n\ s\ of\ \overline{p_k : \mathcal{X}_k}$ is the subterm reduced by applying a *case*-rule in the leftmost-outermost narrowing step, and

   $$t' = \sigma(case_1\ (\ldots(case_{n-1}\ \mathcal{X}_i\ of\ \ldots)\ldots)\ of\ \ldots)$$

   with $\sigma = \{s \mapsto p_i\}$ for some $i \in \{1, \ldots, k\}$. To compute the same result by a LNT-step, we apply the *Case Guess* rule to this goal system:

   $$\mathcal{F}lat(t \to^? X)\ \Rightarrow^\sigma\ \sigma(\mathcal{X}_i \to^? Y_{n-1}, \ldots, case_1\ Y_1\ of\ \ldots \to^? X) =: G$$

   By definition of $\mathcal{F}old$, $\mathcal{F}old(G) = t' \to^? X$.
3. s has a constructor at the top: Then $case_n\ s\ of\ \overline{p_k : \mathcal{X}_k}$ is the subterm reduced by applying a *case*-rule in the leftmost-outermost narrowing step, $\sigma = \{\}$ (since only pattern variables are instantiated), and

   $$t' = case_1\ (\ldots(case_{n-1}\ \varphi(\mathcal{X}_i)\ of\ \ldots)\ldots)\ of\ \ldots$$

   where $p_i = c(\overline{X_n})$ for some $i \in \{1, \ldots, k\}$ and $\varphi = \{\overline{X_n \mapsto t_n}\}$. Since s is

constructor-rooted, we can only apply the *Case Select* rule to the goal system $\mathcal{F}lat(t \to^? X)$:

$$\mathcal{F}lat(t \to^? X) \;\Rightarrow^{\{\}}\; \varphi(\mathcal{X}_i) \to^? Y_{n-1}, \ldots, case_1\ Y_1\ of\ \ldots \to^? X =: G$$

By definition of $\mathcal{F}old$, $\mathcal{F}old(G) = t' \to^? X$.

□

We can extend this lemma to entire leftmost-outermost narrowing derivations.

Lemma A.3
Let $t \leadsto^*_\sigma c$ be a leftmost-outermost narrowing derivation w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$ for the term $t = case\ t_0\ of\ c : c$ and X a fresh variable. Then there exists a LNT-derivation $\mathcal{F}lat(t \to^? X) \stackrel{*}{\Rightarrow}^\sigma c \to^? X$ w.r.t. $\mathcal{R}'$.

Proof
The proof is done by induction on the length n of the leftmost-outermost derivation $t \leadsto^*_\sigma c$.

$n = 1$**:** Then $t \leadsto_\sigma c$. By Lemma A.2, there exists a LNT-step $\mathcal{F}lat(t \to^? X) \Rightarrow^\sigma G$ w.r.t. $\mathcal{R}'$ such that $\mathcal{F}old(G) = c \to^? X$. By definition of $\mathcal{F}old$, $G = c \to^? X$.

$n > 1$**:** Then $t \leadsto^*_\sigma c$ has the form $t \leadsto_\varphi t' \leadsto^{n-1}_\tau c$ with $\sigma = \tau\varphi$. By Lemma A.2, there exists a LNT-step $\mathcal{F}lat(t \to^? X) \Rightarrow^\varphi G$ w.r.t. $\mathcal{R}'$ such that $\mathcal{F}old(G) = t' \to^? X$. By induction hypothesis, there exists a LNT-derivation $\mathcal{F}lat(t' \to^? X) \stackrel{*}{\Rightarrow}^\tau c \to^? X$. If $G = \mathcal{F}lat(t' \to^? X)$, we can join the first LNT-step with this LNT-derivation to the requested LNT-derivation. Hence, consider the case $G \neq \mathcal{F}lat(t' \to^? X)$. Since $\mathcal{F}old(G) = t' \to^? X$, G can be transformed into $\mathcal{F}lat(t' \to^? X)$ by applying *Bind* rules.

□

The following lemma shows that each inference step in the calculus LNT w.r.t. $\mathcal{R}'$ corresponds to zero or one leftmost-outermost narrowing steps w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$.

Lemma A.4
Let t be a term, X a variable which does not occur in t, and $\mathcal{F}lat(t \to^? X) \Rightarrow^\sigma G$ a LNT-step with $G \neq \{\}$. Then, either $\mathcal{F}old(G) = t \to^? X$ or there exists a leftmost-outermost narrowing step $t \leadsto_\sigma t'$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$ such that $\mathcal{F}old(G) = t' \to^? X$.

Proof
Let $\mathcal{F}lat(t \to^? X) = s \to^? Y, F$. We distinguish the different cases for the applied inference rule of the calculus LNT:

1. The *Bind* rule is applied: Then $\sigma = \{\}$ and $G = \varphi(F)$ with $\varphi = \{Y \mapsto s\}$. By definition of $\mathcal{F}old$, $\mathcal{F}old(G) = t \to^? X$.
2. The *Case Select* rule is applied: Then $\sigma = \{\}$, $s = case\ c(\overline{t_n})\ of\ \overline{p_k : \mathcal{X}_k}$, $p_i = c(\overline{X_n})$ for some $i \in \{1, \ldots, k\}$, $\varphi = \{\overline{X_n \mapsto t_n}\}$, and $G = \varphi(\mathcal{X}_i) \to^? Y, F$. By definition of $\mathcal{F}old$, we can apply a leftmost-outermost narrowing step at the position of the subterm s of t with an appropriate *case*-rule, i.e., $t \leadsto_{\{\}} t'$ is a leftmost-outermost narrowing step with $\mathcal{F}old(G) = t' \to^? X$.

3. The *Case Guess* rule is applied: Analogously to the previous case.
4. The *Case Eval* rule is applied: Then $\sigma = \{\}$, $s = case\ f(\overline{t_n})\ of\ \overline{p_k : \mathcal{X}_k}$, $f(\overline{X_n}) \to \mathcal{X}$ is a rewrite rule, $\varphi = \{\overline{X_n \mapsto t_n}\}$, and $G = \varphi(\mathcal{X}) \to^? Z, case\ Z\ of\ \overline{p_k : \mathcal{X}_k}$ for some fresh variable Z. By definition of $\mathcal{F}old$, we can apply a leftmost-outermost narrowing step at the position of the subterm $f(\overline{t_n})$ of t with the same rewrite rule, i.e., $t \leadsto_{\{\}} t'$ is a leftmost-outermost narrowing step with $\mathcal{F}old(G) = t' \to^? X$ (by definition of $\mathcal{F}old$).

□

Now we can prove the equivalence of leftmost-outermost narrowing and the lazy narrowing calculus LNT.

Theorem 3.5
Let t be a term, c a 0-ary constructor, and X a fresh variable. For each leftmost-outermost narrowing derivation $t \leadsto_\sigma^* c$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$ there exists a LNT-derivation $case\ t\ of\ c : c \to^? X \overset{*}{\Rightarrow}^\sigma c \to^? X$ w.r.t. $\mathcal{R}'$, and vice versa.

Proof
First, note that a leftmost-outermost narrowing derivation $t \leadsto_\sigma^* c$ has a unique correspondence to a leftmost-outermost narrowing derivation

$$case\ t\ of\ c : c \leadsto_\sigma^* case\ c\ of\ c : c \leadsto_{\{\}} c\ .$$

The existence of the corresponding LNT-derivation is a direct consequence of Lemma A.3, considering the fact that $\mathcal{F}lat(case\ t\ of\ c : c \to^? X) = case\ t\ of\ c : c \to^? X$ by definition of $\mathcal{F}lat$. The reverse direction follows from Lemma A.4 with a simple induction on the length of the derivations. □