Higher-Order Narrowing with Definitional Trees

Michael Hanus / Christian Prehofer

Aachener Informatik-Bericht
96-2

Higher-Order Narrowing with Definitional Trees*

Michael Hanus† Christian Prehofer‡

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 incomparable solutions. Thus, it is the first calculus for higher-order functional logic programming which provides for such an optimality result. Since we allow higher-order logical variables denoting λ-terms, applications go beyond current functional and logic programming languages.

*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.

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

‡Fakultät für Informatik, Technische Universität München, D-80290 München, Germany, `prehofer@informatik.tu-muenchen.de`

1 Introduction

Functional logic languages [8] with a sound and complete operational semantics are mainly based on narrowing. Narrowing, originally introduced in automated theorem proving [35], is used to *solve* goals by finding appropriate values for variables occurring in arguments of functions. A *narrowing step* instantiates variables in a goal and applies a reduction step to a redex of the instantiated goal. The instantiation of goal variables is usually computed by unifying a subterm of the goal with the left-hand side of some rule.

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

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

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:

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

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

Due to the huge search space of simple narrowing, steadily improved narrowing strategies have been developed in the past. *Needed narrowing* [2] is based on the idea to evaluate only subterms which are *needed* in order to compute some result. For instance, in a goal $t_1 \leq t_2$, it is always necessary to evaluate t_1 (to some head normal form) since all three rules in Example 1.1 have a non-variable first argument. On the other hand, the evaluation of t_2 is only needed if t_1 is of the form $s(\cdots)$. Thus, if t_1 is a free variable, needed narrowing instantiates it to a constructor, here 0 or s. Depending on this instantiation, either the first rule is applied or the second argument t_2 is evaluated. Needed narrowing is the currently best narrowing strategy for first-order functional logic programs due to its optimality properties w.r.t. the length of derivations and the number of computed solutions [2]. Since needed narrowing is defined for *inductively sequential rewrite systems* [1] (these are constructor-based rewrite system 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., [9]).

On ground terms, needed narrowing falls back to the classical notion of "needed reduction" in the sense of Huet and Lévy [13]. Evaluation by needed narrowing has also some similarities to pattern matching and lazy evaluation in functional languages like Haskell [12] or Miranda [29]. 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

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

and a non-terminating function $\perp$. 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 [1].

In this paper, we extend the needed narrowing strategy to higher-order functions and λ-terms as data structures. 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 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 [32, 33, 34]. Furthermore, 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:

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

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. Thus, 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 [33], we show that LNT coincides with needed narrowing in the first-order case. Note that steps of classic narrowing strategies (e.g., [14, 35]) are defined as a variable instantiation followed by the reduction of some *subterm*, whereas more recent lazy narrowing strategies [11, 15, 20, 22] manipulate equation systems in the style of Martelli and Montanari's unification algorithm [19] 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.

- For the higher-order case, we show soundness and completeness with respect to higher-order needed reductions, which we define via definitional trees.

- We show that the calculus is optimal w.r.t. the solutions computed, i.e., no solution is produced twice. Optimality of higher-order reductions is subject of current research. It is however shown that higher-order needed reductions are in fact needed for reduction to a constructor normal form. 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 narrowing strategy if the higher-order features are not used, i.e., our calculus is a conservative extension of an optimal first-order narrowing calculus.

- Since we allow higher-order logical variables denoting λ-terms (similar to λProlog [25] or Escher [17]), applications go beyond current functional and logic programming languages. In general, our calculus can compute solutions for variables of functional type. Although this is very powerful, we show that the incurring higher-order unification can sometimes be avoided by techniques similar to [4].

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. [10]). We assume the following **variable conventions**:

- F, G, H, P, X, Y denote free variables,

- a, b, c, f, g (function) constants, and

- x, y, z bound variables.

Type judgments are written as $t : \tau$. Further, we often use s and t for terms and u, v, w for constants or bound variables. The set of types $\mathcal{T}$ for the simply typed λ-terms is generated by a set $\mathcal{T}_0$ of base types (e.g. int, bool) and the function type constructor $\rightarrow$. The syntax for **λ-terms** is given by

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

A list of syntactic objects $s_1, \ldots, s_n$ where $n \geq 0$ is abbreviated by $\overline{s_n}$. For instance, n-fold abstraction and application are written as $\lambda\overline{x_n}.s = \lambda x_1 \ldots \lambda x_n.s$ and $a(\overline{s_n}) = ((\cdots(a\ s_1)\cdots)\ s_n)$, respectively. **Substitutions** are finite mappings from variables to terms, denoted by $\{\overline{X_n \mapsto t_n}\}$, and extend homomorphically from variables to terms. Free and bound variables of a term t will be denoted as $\mathcal{FV}(t)$ and $\mathcal{BV}(t)$, respectively. A term t is **ground** if $\mathcal{FV}(t) = \{\}$. The **conversions in λ-calculus** are defined as:

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

The long $\beta\eta$-normal form [26] of a term t, denoted by $t\updownarrow_\beta^\eta$, is the η-expanded form of the β-normal form of t. It is well known [10] that $s =_{\alpha\beta\eta} t$ iff $s\updownarrow_\beta^\eta =_\alpha t\updownarrow_\beta^\eta$. As long $\beta\eta$-normal forms exist for typed λ-terms, we will in general assume that terms are in long $\beta\eta$-normal form. For brevity, we may write variables in η-normal form, e.g. X instead of $\lambda\overline{x_n}.X(\overline{x_n})$. We assume that the transformation into long $\beta\eta$-normal form is an implicit operation, e.g. when applying a substitution to a term.

A substitution θ is in long $\beta\eta$-normal form if all terms in the image of θ are in long $\beta\eta$-normal form. The convention that α-equivalent terms are identified and that free and bound variables are kept disjoint (see also [5]) is used in the following. Furthermore, we assume that bound variables with different binders have different names. Define $\mathcal{D}om(\theta) = \{X \mid \theta X \neq X\}$ and $\mathcal{R}ng(\theta) = \bigcup_{X \in \mathcal{D}om(\theta)} \mathcal{FV}(\theta X)$. Two **substitutions are equal on a set of variables W,** written as $\theta =_W \theta'$, if $\theta\alpha = \theta'\alpha$ for all $\alpha \in W$. The restriction of a substitution to a set of variables W is defined as $\theta_{|W}\alpha = \theta\alpha$ if $\alpha \in W$ and $\theta_{|W}\alpha = \alpha$ otherwise. A substitution θ is **idempotent** iff $\theta = \theta\theta$. We will in general assume that substitutions are idempotent. A substitution θ' is more general than θ over a set of variables W, written as $\theta' \leq_W \theta$, if $\theta =_W \sigma\theta'$ for some substitution σ. We describe positions in λ-terms by sequences over natural numbers. The subterm at a **position** p in a λ-term t is denoted by $t|_p$. A term t with the subterm at position p replaced by s is written as $t[s]_p$.

A term t in β-normal form is called a **higher-order pattern** if every free occurrence of a variable F is in a subterm $F(\overline{u_n})$ of t such that the $\overline{u_n}$ are η-equivalent to a list of distinct bound variables. Unification of patterns is decidable and a most general unifier exists if they are unifiable [23]. 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** [26] is a pair $l \rightarrow r$ such that l is a higher-order pattern but not a free variable, l and r are long $\beta\eta$-normal forms of the same base type, and $\mathcal{FV}(l) \supseteq \mathcal{FV}(r)$.

Assuming a rule $l \rightarrow r$ and a position p in a term s in long $\beta\eta$-normal form, a **rewrite step** from s to t is defined as

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

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

By applying rewrite steps, we can compute the *value* of a functional expression. However, in the presence of free variables, we have to compute values for these free variables such that the instantiated expression is reducible. This is the motivation for narrowing which will be precisely defined in the following sections. Narrowing is intended to *solve* goals, where a **goal** is an expression of Boolean type that should be reduced to the constant $true$. This is general enough to cover the equation solving capabilities of current functional logic languages with a lazy operational semantics, like BABEL [24] or K-LEAF [6], since the strict equality $\approx$ can be defined as a binary operation by a set of orthogonal rewrite rules.

We follow the same approach and interpret equations as terms by defining the symbol $\approx$ as a binary operation (more precisely, one operation for each base type). The operation $\approx$ is defined by 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.

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

With these rules an equation is valid if it can be rewritten to $true$ (see [2, 6, 24] for more details about strict equality).[1] This interpretation of equality in goals is also taken in functional logic languages with higher-order features [7].

The substitution σ is a **solution** of a goal G iff $\sigma(G)$ can be rewritten to $true$. An important consequence of this restriction on goals is the fact that during the successful rewriting of a goal the topmost symbol is always an operation or the constant $true$. This property will be used to simplify the narrowing calculus.

Notice that a subterm $s|_p$ may contain free variables which used to be bound in s. For rewriting it is possible to ignore this, as only matching of a left-hand side of a rewrite rule is needed. For narrowing, we need unification and hence we use the following construction to lift a rule into a binding context to facilitate the technical treatment. An **$\overline{x_k}$-lifter** of a term t **away from** W is a substitution $\sigma = \{F \mapsto (\rho F)(\overline{x_k}) \mid F \in \mathcal{FV}(t)\}$ where ρ is a renaming such that $\mathcal{D}om(\rho) = \mathcal{FV}(t)$, $\mathcal{R}ng(\rho) \cap W = \{\}$ and $\rho F : \tau_1 \rightarrow \cdots \rightarrow \tau_k \rightarrow \tau$ if $x_1 : \tau_1, \ldots, x_k : \tau_k$ and $F : \tau$. A term t (rewrite rule $l \rightarrow r$) is $\overline{x_k}$-lifted if an $\overline{x_k}$-lifter has

[1] Note that normal forms may not exist in general due to non-terminating rewrite rules.

been applied to t (l and r). For example, $\{G \mapsto G'(x)\}$ is an x-lifter of $g(G)$ away from any W not containing G'.

3 First-Order Definitional Trees

Definitional trees are introduced in [1] to define efficient normalization strategies for (first-order) term rewriting. The idea is to represent all rules for a defined symbol in a tree and to control the selection of the next redex by this tree. This technique is extended to narrowing in [2] 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 and functional variables do not occur.

A traditional narrowing step [8] 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 $t \leadsto_{\sigma|_{\mathcal{FV}(t)}} t'$.[2] 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 [8] for a survey). A **narrowing strategy** determines the position where the next narrowing step should be applied. As shown in [2], an optimal narrowing strategy can be obtained by dropping the requirement for most general unifiers and controlling the instantiation of variables and selection of narrowing positions by a data structure, called definitional tree. To provide a precise definition of this needed narrowing strategy, we first recall the notion of a definitional tree. $\mathcal{T}$ is a **definitional tree** with pattern π iff its depth is finite and one of the following cases holds:

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

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

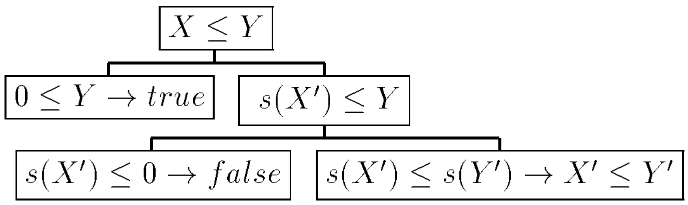
We denote by $pat(\mathcal{T})$ the **pattern** of the definitional tree $\mathcal{T}$. A **definitional tree** of an n-ary function f is a definitional tree $\mathcal{T}$ with pattern $f(\overline{X_n})$, where $\overline{X_n}$ are distinct variables, such that for each rule $l \to r$ with $l = f(\overline{t_n})$ there is a node $rule(l' \to r')$ in $\mathcal{T}$ with l variant

[2] 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.

of l'.[3] For instance, the rules in Example 1.1 can be represented by the following definitional tree:

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

This tree can be illustrated by the following picture:



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

A function f is called **inductively sequential** if there exists a definitional tree of f such that each *rule* node corresponds to exactly one rule of the rewrite system $\mathcal{R}$. The term rewriting system $\mathcal{R}$ is called inductively sequential if each function defined by $\mathcal{R}$ is inductively sequential. Thus, inductively sequential rewrite rules are a subclass of constructor-based orthogonal rewrite systems which are appropriate to reflect current functional languages.

3.1 Narrowing with Definitional Trees

A definitional tree defines a strategy to apply narrowing steps. To narrow a term t, we consider the definitional tree $\mathcal{T}$ of the outermost function symbol of t (note that, by our restriction on goals, the outermost symbol is always a Boolean function).

$\mathcal{T} = rule(l \rightarrow r)$: 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 head normal form) 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 $\mathcal{T}_j$ unifies with t, otherwise (if no pattern unifies) 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 $\mathcal{T}_j$.

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

This strategy is called **needed narrowing** [2] and is the currently best narrowing strategy due to its optimality w.r.t. the length of derivations (if terms are shared) and the number of computed solutions.

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.[4] 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.[5] 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.

$$
\begin{array}{l}
s(X) \leq Y \\
\qquad \Rightarrow_{Initial} \\
\mathcal{E}val(s(X) \leq Y, branch(X_1 \leq Y_1, 1, rule(\cdots), branch(s(X_2) \leq Y_1, \ldots))) \\
\qquad \Rightarrow_{Select} \\
\mathcal{E}val(s(X) \leq Y, branch(s(X_2) \leq Y_1, 2, rule(s(X_2) \leq 0 \to \cdots), rule(s(X_2) \leq s(Y_2) \to \cdots))) \\
\qquad \Rightarrow^{\{Y \mapsto s(Y_2)\}}_{Instantiate} \\
\mathcal{E}val(s(X) \leq s(Y_2), rule(s(X_2) \leq s(Y_2) \to X_2 \leq Y_2)) \\
\qquad \Rightarrow_{Apply} \\
X \leq Y_2 \\
\qquad \Rightarrow_{Initial} \\
\mathcal{E}val(X \leq Y_2, branch(X_3 \leq Y_3, 1, rule(0 \leq Y_3 \to true), branch(s(X_4) \leq Y_3, \ldots))) \\
\qquad \Rightarrow^{\{X \mapsto 0\}}_{Instantiate} \\
\mathcal{E}val(0 \leq Y_2, rule(0 \leq Y_3 \to true)) \quad \Rightarrow_{Apply} \quad true
\end{array}
$$

[4] This description of needed narrowing is slightly different than in [2] but more appropriate for the subsequent proofs. In [2], 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.

[5] As in proof procedures for logic programming, we assume that we take a definitional tree with fresh variables in each such evaluation step.

Initial

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

Apply

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

Select

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

Instantiate

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

Eval Subterm

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

Replace Subterm

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

Figure 1: Calculus for needed narrowing

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 [30] 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:

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

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):

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

To be more precise, we translate a definitional tree $\mathcal{T}$ into a term with case expressions by the use of the translation function $\mathcal{C}ase(\mathcal{T})$:

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

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

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

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 $2n + 1$ where the semantics is defined by the following n rewrite rules:[6]

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

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.

[6]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.

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 $\mathcal{R}_c$ 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.[7]

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

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

To relate needed narrowing derivations and leftmost-outermost derivations with case expressions, we define the following translation $\mathcal{EC}$ from $\mathcal{E}val$-goals into terms with case expressions.

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

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

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

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

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. $\mathcal{R}' \cup \mathcal{R}_c$.

Lemma 3.3 *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 $t \rightsquigarrow_\sigma t'$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$ 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 a definitional tree for f with pattern $f(\overline{X_n})$ and $\varphi = \{\overline{X_n \mapsto t_n}\}$. Then $t \rightsquigarrow_{\{\}} \varphi(\mathcal{C}ase(\mathcal{T}))$ is a unique leftmost-outermost narrowing step w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$. 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., $t \rightsquigarrow_{\{\}} \varphi'(\varphi(\mathcal{C}ase(\mathcal{T}_i)))$ 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 a 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,

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

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

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

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

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

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 a 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. $\mathcal{R}' \cup \mathcal{R}_c$ is based on the previous lemma:

Bind

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

Case Select

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

Case Guess

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

Case Eval

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

Figure 2: Calculus LNT for lazy narrowing with definitional trees in the first-order case

Theorem 3.4 *Let t be a term with a Boolean function at the top. For each needed narrowing derivation $t \leadsto^*_\sigma true$ w.r.t. $\mathcal{R}$ there exists a leftmost-outermost narrowing derivation $t \leadsto^*_\sigma true$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$, and vice versa.*

Proof By induction on the derivation steps and applying Lemma 3.3, 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 3.3) that each leftmost-outermost narrowing derivation $t \leadsto^*_\sigma true$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$ corresponds to a needed narrowing derivation $t \leadsto^*_\sigma true$ w.r.t. $\mathcal{R}$. $\square$

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 this purpose, we transform a leftmost-outermost narrowing derivation w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$ into a derivation on a **goal system** G (a sequence of goals of the form $t \to^? X$) where narrowing rules are only applied to the outermost function symbol of the leftmost goal. This is the purpose of the inference system LNT shown in Figure 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 solving goals by reduction to $true$, we assume that the **initial goal** $\mathcal{I}(t)$ for a term t has always the form $case\ t\ of\ true : true \to^? T$. We use this representation in order to provide a calculus with fewer inference rules. Note that $T \mapsto true$ if such a goal can be reduced to the empty goal system.

Example 3.5 The following LNT-derivation corresponds to the leftmost-outermost narrowing derivation shown in Example 3.2.

$$
\begin{array}{ll}
\multicolumn{2}{l}{case\ s(X) \leq Y\ of\ true : true \rightarrow^? T} \\
\Rightarrow_{Case\ Eval} & case\ s(X)\ of\ 0 : true,\ s(X_1) : (case\ Y\ of\ 0 : false,\ s(Y_1) : X_1 \leq Y_1) \rightarrow^? Z, \\
& case\ Z\ of\ true : true \rightarrow^? T \\
\Rightarrow_{Case\ Select} & case\ Y\ of\ 0 : false,\ s(Y_1) : X \leq Y_1 \rightarrow^? Z,\ case\ Z\ of\ true : true \rightarrow^? T \\
\Rightarrow^{\{Y \mapsto s(Y_1)\}}_{Case\ Guess} & X \leq Y_1 \rightarrow^? Z,\ case\ Z\ of\ true : true \rightarrow^? T \\
\Rightarrow_{Bind} & case\ X \leq Y_1\ of\ true : true \rightarrow^? T \\
\Rightarrow_{Case\ Eval} & case\ X\ of\ 0 : true,\ s(X_2) : (case\ Y_1\ of\ 0 : false,\ s(Y_2) : X_2 \leq Y_2) \rightarrow^? Z', \\
& case\ Z'\ of\ true : true \rightarrow^? T \\
\Rightarrow^{\{X \mapsto 0\}}_{Case\ Guess} & true \rightarrow^? Z',\ case\ Z'\ of\ true : true \rightarrow^? T \\
\Rightarrow_{Bind} & case\ true\ of\ true : true \rightarrow^? T \\
\Rightarrow_{Case\ Select} & true \rightarrow^? T \\
\Rightarrow_{Bind} & \{\}
\end{array}
$$

The inference rules LNT of the calculus LNT keep the following important invariant on goal systems:

(∗) If $G_1, l \rightarrow^? r, G_2$ is a goal system, then r is a variable which does not occur in G_1 and l.

There is a strong correspondence between terms with case expressions and goal systems, since each single equation $t \rightarrow^? X$ can be "flattened" into a goal system by the following function $\mathcal{F}lat$:[8]

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

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

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

we obtain the "flattened" goal system

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

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:

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

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

[8]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.

Lemma 3.6 *Let $t \leadsto_\sigma t'$ be a leftmost-outermost narrowing step in a derivation of the initial term $case\ t_0\ of\ true : true$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$ 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 $case\ t_0\ of\ true : true$, t has a $case$ symbol at the top. Moreover, since it was derived by leftmost-outermost narrowing steps w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$, 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 $\mathcal{F}lat(t \to^? X)$:

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

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

$\square$

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

Lemma 3.7 *Let $t = case\ t_0\ of\ true : true$, $t \rightsquigarrow^*_\sigma true$ be a leftmost-outermost narrowing derivation w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$, and X a fresh variable. Then there exists a LNT-derivation $\mathcal{F}lat(t \rightarrow^? X) \stackrel{*}{\Rightarrow}^\sigma true \rightarrow^? X$ w.r.t. $\mathcal{R}'$.*

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

$n = 1$: Then $t \rightsquigarrow_\sigma true$. By Lemma 3.6, there exists a LNT-step $\mathcal{F}lat(t \rightarrow^? X) \Rightarrow^\sigma G$ w.r.t. $\mathcal{R}'$ such that $\mathcal{F}old(G) = true \rightarrow^? X$. By definition of $\mathcal{F}old$, $G = true \rightarrow^? X$.

$n > 1$: Then $t \rightsquigarrow^*_\sigma true$ has the form $t \rightsquigarrow_\varphi t' \rightsquigarrow^{n-1}_\tau true$ with $\sigma = \tau\varphi$. By Lemma 3.6, there exists a LNT-step $\mathcal{F}lat(t \rightarrow^? X) \Rightarrow^\varphi G$ w.r.t. $\mathcal{R}'$ such that $\mathcal{F}old(G) = t' \rightarrow^? X$. By induction hypothesis, there exists a LNT-derivation $\mathcal{F}lat(t' \rightarrow^? X) \stackrel{*}{\Rightarrow}^\tau true \rightarrow^? X$. If $G = \mathcal{F}lat(t' \rightarrow^? X)$, we can join the first LNT-step with this LNT-derivation to the requested LNT-derivation. Hence, consider the case $G \neq \mathcal{F}lat(t' \rightarrow^? X)$. Since $\mathcal{F}old(G) = t' \rightarrow^? X$, G can be transformed into $\mathcal{F}lat(t' \rightarrow^? X)$ by applying *Bind* rules. $\square$

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. $\mathcal{R}' \cup \mathcal{R}_c$.

Lemma 3.8 *Let t be a term, X a variable which does not occur in t, and $\mathcal{F}lat(t \rightarrow^? X) \Rightarrow^\sigma G$ a LNT-step with $G \neq \{\}$. Then, either $\mathcal{F}old(G) = t \rightarrow^? X$ or there exists a leftmost-outermost narrowing step $t \rightsquigarrow_\sigma t'$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$ such that $\mathcal{F}old(G) = t' \rightarrow^? X$.*

Proof Let $\mathcal{F}lat(t \rightarrow^? X) = s \rightarrow^? Y, F$. 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 $\mathcal{F}old$, $\mathcal{F}old(G) = t \rightarrow^? X$.

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 $G = \varphi(\mathcal{X}_i) \rightarrow^? Y, F$. 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., $t \rightsquigarrow_{\{\}} t'$ is a leftmost-outermost narrowing step with $\mathcal{F}old(G) = t' \rightarrow^? X$.

3. The *Case Guess* rule is applied: Analogously to the previous case.

4. The *Case Eval* rule is applied: Then $\sigma = \{\}$, $s = case\ f(\overline{t_n})\ of\ \overline{p_k : \mathcal{X}_k}$, $f(\overline{X_n}) \rightarrow \mathcal{X}$ is a rewrite rule, $\varphi = \{\overline{X_n \mapsto t_n}\}$, and $G = \varphi(\mathcal{X}) \rightarrow^? Z, case\ Z\ of\ \overline{p_k : \mathcal{X}_k}$ 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., $t \rightsquigarrow_{\{\}} t'$ is a leftmost-outermost narrowing step with $\mathcal{F}old(G) = t' \rightarrow^? X$ (by definition of $\mathcal{F}old$).

$\square$

The following theorem states the equivalence of leftmost-outermost narrowing and the lazy narrowing calculus LNT.

Theorem 3.9 *Let t be a term with a Boolean function at the top and X a fresh variable. For each leftmost-outermost narrowing derivation $t \leadsto^*_\sigma true$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_c$ there exists a LNT-derivation $case\ t\ of\ true : true \rightarrow^? X \quad \stackrel{*}{\Rightarrow}^\sigma \quad true \rightarrow^? X$ w.r.t. $\mathcal{R}'$, and vice versa.*

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

$$case\ t\ of\ true : true \ \leadsto^*_\sigma\ case\ true\ of\ true : true\ \leadsto_{\{\}}\ true\ .$$

The existence of the corresponding LNT-derivation is a direct consequence of Lemma 3.7, considering the fact that $\mathcal{F}lat(case\ t\ of\ true : true \rightarrow^? X) = case\ t\ of\ true : true \rightarrow^? X$ by definition of $\mathcal{F}lat$. The reverse direction follows from Lemma 3.8 with a simple induction on the length of the derivations. $\square$

Theorems 3.4 and 3.9 imply the equivalence of needed narrowing and the calculus LNT. Since we will extend LNT to higher-order functions in the next section, the results in this section show that our higher-order calculus is a conservative extension of an optimal first-order narrowing strategy.

4 Higher-Order Definitional Trees

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

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 node $\mathcal{T}'$ in a tree $\mathcal{T}$, the constraints in the case expressions on the path to it determine a term, which is recursively defined by the pattern function $pat_{\mathcal{T}}(\mathcal{T}')$:

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

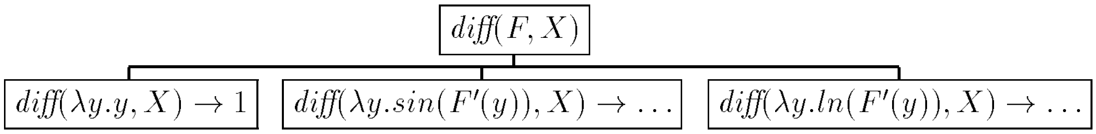
Each branch variable must belong to the pattern of this node, i.e., for each node $\mathcal{T}' = p : case\ X\ of\ \overline{\mathcal{T}_n}$ in a tree $\mathcal{T}$, X is a free variable of $pat_{\mathcal{T}}(\mathcal{T}')$. Furthermore, each leaf $\mathcal{T}' = p : rhs$ of a *hdt* $\mathcal{T}$ is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

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

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



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 [28], 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 [31, 33]. 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.

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. The Imitation and Projection rules are taken from higher-order unification and compute a partial binding for some variable. The Function Guess rule covers the case of non-constructor solutions, which may occur for higher-order variables. It thus enables the synthesis of functions from existing ones. Note that the selection of a binding in this rule is only restricted by the types occurring. For all rules, we assume that newly introduced variables are fresh, as in the first-order case.

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

For a sequence $\Rightarrow^{\theta_1} \cdots \Rightarrow^{\theta_n}$ of LNT steps, we write $\stackrel{*}{\Rightarrow}{}^{\theta}$, where $\theta = \theta_n \cdots \theta_1$. 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 [31].

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

Example 5.1 As an example, consider the goal

$$\lambda x.diff(\lambda y.sin(F(x,y)), x) \rightarrow^? \lambda x.cos(x)$$

w.r.t. the rules for *diff* (see Example 1.2) and the *hdt* for the function $*$:

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

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

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 $p_i = \lambda\overline{y_l}.c(\overline{X_o(\overline{x_k}, \overline{y_l})})$ and $\sigma = \{X \mapsto \lambda\overline{x_m}.c(\overline{H_o(\overline{x_m})})\}$

Function Guess

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

Projection

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

Case Eval

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

Figure 3: System LNT for needed narrowing in the higher-order case

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

$\Rightarrow_{Case\ Eval}$

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

$\Rightarrow_{Case\ Eval}$

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

$\Rightarrow_{Case\ Select}$

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

$\Rightarrow_{Bind}$

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

$\Rightarrow_{Case\ Eval}$

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

$\Rightarrow_{Case\ Eval}$

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

$\Rightarrow^{\{F \mapsto \lambda x,y.y\}}_{Projection}$

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

$\Rightarrow_{Case\ Select}$

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

$\Rightarrow_{Bind}$

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

$\Rightarrow_{Case\ Select}$

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

$\Rightarrow_{Bind}$

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

$\Rightarrow_{Case\ Select}$

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

$\Rightarrow_{Bind}$

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

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

6 Correctness and Completeness

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

6.1 Needed Reductions

For our purpose it is convenient to define needed reductions via LNT. Then we show that they are in fact needed. For modeling rewriting, the Guess rules are not needed: $S \overset{*}{\Rightarrow}{}^{\{\}}_{LNT} S'$ 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 **labeled 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 labeled.

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

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

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

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

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

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 often omitted in goal systems for brevity.

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 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 $S = case_\epsilon\ t\ of\ true : true \to^? X$, we define the associated substitution $\theta_S = \{X \mapsto t\}$.

- For the Case Eval rule on $S = \lambda\overline{x}.case_p\ \lambda\overline{y}.f(\overline{t})\ of \ldots \to^? X, G$ with

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

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

For all other rules, the associated substitution is unchanged.

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

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

Definition 6.3 For a goal system S of the form

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

(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 $S_1 = case_\epsilon\ t\ of\ true : true \to^? X$, then $\mathcal{A}(S_1) = t$.

For a goal system S, we write $S\!\downarrow$ for the normal form obtained by applying Case Eval and Case Select. 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 in the leftmost case construct decreases. 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

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

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:

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

Since $\theta_S X|_p = v(\overline{t_n})$ and $v(\overline{t_n})|_{p'} = t_i$, $\theta_{S'} X|_{p \cdot p'} = t_i$ 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 $\mathcal{A}(\mathcal{I}(t)) = \mathcal{A}(\mathcal{I}(t)\!\downarrow)$.

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

Stability of reduction under substitution can be shown since needed reductions are outermost.

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

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

The next goal is to relate a rewrite step with LNT computations.

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

$$\begin{array}{ll}
\multicolumn{2}{l}{\lambda \overline{x}.case\ \lambda \overline{y}.l\ of\ \overline{\mathcal{T}_m} \rightarrow^? X, G} \\
\quad \Rightarrow_{Case\ Eval} \overset{*}{\Rightarrow}_{Case\ Select} & \lambda \overline{x}.case\ \lambda \overline{y}.p_i\ of\ \overline{p_n : \mathcal{X}_n} \rightarrow^? X', \\
& \qquad \lambda \overline{x}.case\ \lambda \overline{y}.X'(\overline{x},\overline{y})\ of\ \overline{\mathcal{T}_m} \rightarrow^? X, G \\
\qquad \Rightarrow_{Case\ Select} & \lambda \overline{x}.\sigma(\mathcal{X}_i) \rightarrow^? X', \lambda \overline{x}.case\ \lambda \overline{y}.X'(\overline{x},\overline{y})\ of\ \overline{\mathcal{T}_m} \rightarrow^? X, G
\end{array}$$

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 $\mathcal{T}$. □

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

$$\begin{array}{ll}
\multicolumn{2}{l}{\lambda \overline{x}.case\ \lambda \overline{y}.l\ of\ \overline{\mathcal{T}_n} \rightarrow^? X, G} \\
\quad \Rightarrow_{Case\ Eval} \overset{*}{\Rightarrow}_{Case\ Select} & \lambda \overline{x},\overline{y}.\sigma r \rightarrow^? X', \lambda \overline{x}.case\ \lambda \overline{y}.X'(\overline{x},\overline{y})\ of\ \overline{\mathcal{T}_n} \rightarrow^? X, G
\end{array}$$

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 Case Bind rule. Notice that the substitution of the Bind rule only affects the two leftmost goals.

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

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

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

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

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

Since each path of a definitional tree corresponds to a left-hand side, $t|_p$ is reducible by Corollary 6.9. To show $\mathcal{I}(t'){\downarrow} = Bind(S{\downarrow}){\downarrow}$, consider the computation for $\mathcal{I}(t'){\downarrow}$. Since t

and t' differ only at position p, i.e., $t' = t[\sigma r']_p$ for some rule $l' \to r'$, $\mathcal{I}(t')$ is transformed by LNT to

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

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, $Bind(S\!\downarrow)\!\downarrow = Bind(S'')\!\downarrow = S_1\!\downarrow = \mathcal{I}(t')\!\downarrow$ follows. $\Box$

Theorem 6.11 *For a term t, $\mathcal{I}(t) \stackrel{*}{\Rightarrow}{}^{\{\}}_{LNT} \{\}$ iff $t \stackrel{*}{\longrightarrow} true$.*

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

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

Now, we can define needed reductions:

Definition 6.12 A term t has a needed redex p if $\mathcal{I}(t)\!\downarrow$ is of Invariant 2 with $p = p_1 \cdots p_n$.

It remains to show that needed reductions are indeed needed to compute a constructor headed term. For a normalization result for the first-order case, we refer to [21].

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

Proof The first claim, that t has a needed redex at p, follows from Lemma 6.10.

We show that t must be reduced at p (or is not reducible to true) by induction on the computation of $\mathcal{I}(t)\!\downarrow$. We argue that for a goal system of the form

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

the leftmost case expression must either be reducible by Case Select or s must have a needed redex (to be reducible to a constructor-headed term). In the latter case, s is rooted by a defined symbol and s must be evaluated to a constructor-headed term. Otherwise, no reduction of t to $true$ is possible. $\Box$

The next desirable result is to show that needed reductions are normalizing. This is suggested from related works [27, 16], but is beyond the scope of this paper.

6.2 Lifting Rewriting to Narrowing

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

Lemma 6.14 *If $\mathcal{I}(t) \stackrel{*}{\Rightarrow}{}^{\theta}_{LNT} S \Rightarrow^{\theta'}_{LNT} S'$, 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 $\stackrel{*}{\Rightarrow}{}^{\theta}_{LNT}$. $\Box$

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

Proof by induction on the length of $\stackrel{*}{\Rightarrow}_{LNT}$. Assume $S \Rightarrow^{\theta'} S' \stackrel{*}{\Rightarrow}{}^{\theta''} \{\}$. By induction hypothesis $\theta'' S' \stackrel{*}{\Rightarrow}{}^{\{\}}_{LNT} \{\}$. First, we show $\theta' S \stackrel{*}{\Rightarrow} S'$ 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 $\theta' S \stackrel{*}{\Rightarrow}{}^{\{\}} S'$, $\theta'' S' \stackrel{*}{\Rightarrow}{}^{\{\}} \{\}$, and infer $\theta''\theta' S \stackrel{*}{\Rightarrow}{}^{\{\}} \theta'' S' \stackrel{*}{\Rightarrow}{}^{\{\}} \{\}$ from Lemma 6.7. □

Theorem 6.16 (Correctness of LNT) *If* $\mathcal{I}(t) \stackrel{*}{\Rightarrow}{}^{\theta}_{LNT} \{\}$ *for a term* t, *then* $\theta t \stackrel{*}{\longrightarrow} true$.

Proof First, $\mathcal{I}(\theta t) \stackrel{*}{\Rightarrow}_{LNT} \{\}$ follows from Lemma 6.15 and $\theta t \stackrel{*}{\longrightarrow} true$ from Theorem 6.11. □

We first state completeness in terms of LNT reductions.

Lemma 6.17 *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 $\mathcal{D}om(\theta)$.

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 [36, 33]). 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 it is to 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 [33]) and hence $\theta S \stackrel{*}{\Rightarrow}{}^{\{\}}_{LNT} \{\}$ is impossible.

□

Theorem 6.18 (Completeness of LNT) *If $\theta t \xrightarrow{*} true$ and θ is in $\mathcal{R}$-normal form, then $\mathcal{I}(t) \overset{*}{\Rightarrow}{}^{\theta'}_{LNT} \{\}$ with $\theta' \leq_{\mathcal{FV}(t)} \theta$.*

Proof Completeness follows from Theorem 6.11 and the previous lemma. $\square$

7 Optimality regarding Solutions

We show here another important aspect, namely uniqueness of the computed solutions. Compared to the more general case in [33], 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.

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

Proof The claim follows from examining the substitutions computed. First, it is to observe that, except for the case rules, no rule overlap, i.e., apply simultaneously. At each choice point for a Guess rule, the rule compute distinct bindings:

- $\lambda \overline{x_n}.c(\overline{H_m(\overline{x_n})})$ (Case Guess)
- $\lambda \overline{x_n}.f(\overline{H_m(\overline{x_n})})$ (Function Guess)
- $\lambda \overline{x_n}.x_i(\overline{H_m(\overline{x_n})})$ (Projection)

These bindings occur in the solution for the original term, which is easy to see via Lemma 6.14. Furthermore, the only other bindings computed are for the intermediate variables on the right, where no branching is needed. These bindings do not occur in the computed solution. $\square$

It is also conjectured that our notion of needed reductions is optimal (this is subject to current research [3, 27, 28]). Note, however, that sharing is needed for optimality, as shown for the first-order case in [2].

8 Avoiding Function Synthesis

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

We show that the corresponding result in [4] is easy to see in our context, although lifting over binders obscures the results somewhat unnecessarily.[9] Lifting may instantiate a first-order variable by a higher-order one, but this is only needed to handle the context correctly.

[9]Considering liftings is missing in [18]

A term t is **quasi first-order** if t is a higher-order pattern without free higher-order variables. A rule $f(\overline{X_n}) \rightarrow \mathcal{X}$ 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

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

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

Theorem 8.1 *If $\mathcal{I}(t) \stackrel{*}{\Rightarrow}_{LNT} S$ 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 $f(\overline{X_n}) \rightarrow \mathcal{X}$ 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.

9 Conclusions

We have presented an effective model for the integration of functional and logic programming with completeness and optimality results. Since we do not require terminating rewrite rules and permit higher-order logical variables and λ-abstractions, our strategy is a suitable basis for truly higher-order functional logic languages. Moreover, our strategy reduces to an optimal first-order strategy if the higher-order features are not used. Further work will focus on adapting the explicit model for sharing using goal systems from [33] to this refined context.

References

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

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

[3] Andrea Asperti and Cosimo Laneve. Interaction systems I: The theory of optimal reductions. *Mathematical Structures in Computer Science*, 4:457–504, 1994.

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

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

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

[7] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. On the completeness of narrowing as the operational semantics of functional logic programming. In *Proc. CSL'92*, pages 216–230. Springer LNCS 702, 1992.

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

[9] M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation*, pages 252–266. Springer LNCS 1048, 1995.

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

[11] S. Hölldobler. *Foundations of Equational Logic Programming.* Springer LNCS 353, 1989.

[12] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

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

[14] J.-M. Hullot. Canonical forms and unification. In *Proc. 5th Conference on Automated Deduction*, pages 318–334. Springer LNCS 87, 1980.

[15] T. Ida and K. Nakahara. Leftmost outside-in narrowing calculi. Technical report ISE-TR-94-107, University of Tsukuba, 1994. To appear in Journal of Functional Programming.

[16] Jan Willem Klop. *Combinatory Reduction Systems.* Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.

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

[18] C. A. Loría-Sáenz. *A Theoretical Framework for Reasoning about Program Construction Based on Extensions of Rewrite Systems.* PhD thesis, Univ. Kaiserslautern, December 1993.

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

[20] A. Martelli, G.F. Rossi, and C. Moiso. Lazy unification algorithms for canonical rewrite systems. In Hassan Aït-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures, Volume 2, Rewriting Techniques*, chapter 8, pages 245–274. Academic Press, New York, 1989.

[21] Aart Middeldorp. Call by need computations to root-stable form. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 94–105, January 1997.

[22] Aart Middeldorp, Satoshi Okui, and Tetsuo Ida. Lazy narrowing: Strong completeness and eager variable elimination. In *Proceedings of the 20th Colloquium on Trees in Algebra and Programming.* Springer LNCS, 1995. Full version appeared as Report ISE-TR-95-119, University of Tsukuba, 1995.

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

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

[25] Gopalan Nadathur and Dale Miller. Higher-order logic programming. Technical Report CS-1994-38, Department of Computer Science, Duke University, December 1994. To appear in *Volume 5* of *Handbook of Logic in Artificial Intelligence and Logic Programming*, D. Gabbay, C. Hogger and A. Robinson (eds.), Oxford University Press.

[26] Tobias Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349, 1991.

[27] Vincent van Oostrom. *Confluence for Abstract and Higher-Order Rewriting.* PhD thesis, Vrije Universiteit, 1994. Amsterdam.

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

[29] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[30] Christian Prehofer. Higher-order narrowing. In *Proc. Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 507–516. IEEE Computer Society Press, 1994.

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

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

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

[34] Christian Prehofer. Some applications of functional logic programming. In *Proc. JICSLP'96 Workshop on Multi-Paradigm Logic Programming*, pages 35–45. TU Berlin, Technical Report No. 96-28, 1996.

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

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