Preliminary version. Final version in JACM 47(4):776-822, 2000

A Needed Narrowing Strategy

SERGIO ANTOY

Portland State University, Portland, Oregon

RACHID ECHAHED

Laboratoire LEIBNIZ, Institut IMAG, Grenoble, France

MICHAEL HANUS

Christian-Albrechts-Universität zu Kiel, Germany

Abstract: The narrowing relation over terms constitutes the basis of the most important operational semantics of languages that integrate functional and logic programming paradigms. It also plays an important role in the definition of some algorithms of unification modulo equational theories which are defined by confluent term rewriting systems. Due to the inefficiency of simple narrowing, many refined narrowing strategies have been proposed in the last decade. This paper presents a new narrowing strategy which is optimal in several respects. For this purpose we propose a notion of a needed narrowing step that, for inductively sequential rewrite systems, extends the Huet and Lévy notion of a needed reduction step. We define a strategy, based on this notion, that computes only needed narrowing steps. Our strategy is sound and complete for a large class of rewrite systems, is optimal w.r.t. the cost measure that counts the number of distinct steps of a derivation, computes only incomparable and disjoint unifiers, and is efficiently implemented by unification.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.1.6 [**Programming Techniques**]: Logic Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures*; D.3.4 [**Programming Languages**]: Processors—*Optimization*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Trees*; I.1.1 [**Algebraic Manipulation**]: Expressions and Their Representation—*Simplification of expressions.*

General Terms: Algorithms, Languages, Performance, Theory.

Additional Key Words: Functional Logic Programming Languages, Rewrite Systems, Narrowing Strategies, Call-By-Need.

A preliminary version of this paper appeared in the *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, Oregon, January 1994.

The Oregon Center for Advanced Technology Education (OCATE) supported in part the collaborative efforts that lead to the writing of this paper.

Sergio Antoy was supported in part by the National Science Foundation grants CCR-9406751.

Michael Hanus was supported in part by the German Ministry for Research and Technology (BMFT) under grant ITS 9103, by the ESPRIT Basic Research Working Group 6028 (Construction of Computational Logics), and by the German Research Council (DFG) under grant Ha 2457/1-1.

Authors' addresses: Sergio Antoy, Department of Computer Science, Portland State University, P.O. Box 751, Portland, OR 97207, U.S.A., `antoy@cs.pdx.edu`; Rachid Echahed, IMAG-Leibniz, CNRS, 46, avenue Félix Viallet, F-38031 Grenoble, France, `echahed@imag.fr`; Michael Hanus, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, D-24098 Kiel, Germany, `mh@informatik.uni-kiel.de`.

1 Introduction

Declarative programs are more abstract than equivalent imperative programs. Declarative languages replace pointers with algebraic data types, split complex computations into small, easily parameterizable units and avoid the manipulation of an explicit state through assignments. These features promise to ease some difficult essential tasks of software development. For example, they simplify reasoning about programs (verification of non-executable specifications), promote freedom of implementation (use of parallel architectures), and reduce both development time and maintenance efforts (code is compact and easier to read and understand). All these advantages stem from various factors—the most important being the solid mathematical foundations of declarative computing paradigms.

Currently, the field of declarative programming is split into two main paradigms based on different mathematical formalisms: *functional programming* (lambda calculus) and *logic programming* (predicate logic). This situation has a negative impact on teaching, research and applications. Usually there are different courses on functional programming and logic programming, and students do not perceive the similarities between them. In terms of research, each field has its own community, conferences, and journals, and sometimes similar solutions are developed twice. Each field also has its own application areas and some effort has been devoted to show that one paradigm can cover applications of the other paradigm [71] instead of showing the advantages of declarative programming in various application fields.

Each paradigm, of course, has its own advantages. Functional programming offers nested expressions, efficient evaluation by deterministic (often lazy) evaluation, and higher-order functions. Logic programming offers existentially quantified variables, partial data structures, and built-in search. On the other hand, functional and logic languages have a common core and can be seen as different facets of a single idea. Consequently, the interest in integrating functional and logic programming has grown over the last decade and resulted in various proposals of integrated *functional logic* languages that combine the advantages of both paradigms (see [31] for a survey). Functional logic languages extend both functional languages and logic languages. Functional languages are extended with facilities such as function inversion, partial data structures, and logic variables [65]. Logic languages are extended with nested expressions, a more efficient operational behavior [30], and less need for impure control features such as the Prolog "cut."

This paper concerns narrowing. *Narrowing* is a computation model of considerable importance both for declarative programming in general and for functional logic languages in particular. We explain why using an example.

Example 1 Consider the following rules defining the concatenation of lists (as an infix operator ++) where we use the Prolog syntax for lists, i.e., [] denotes the empty list and $[E|R]$ denotes a non-empty list consisting of a first element E and a remaining list R:

$$\begin{array}{rcl} [\,] \mathbin{+\!\!+} L & \to & L \\ [E|R] \mathbin{+\!\!+} L & \to & [E|R \mathbin{+\!\!+} L] \end{array}$$

In a functional language, this definition is used to concatenate two lists, e.g., $[a, b]$ ++$[c, d]$ evaluates to the list $[a, b, c, d]$. It is understood that (the value of) the arguments of ++ must be known in order to apply a rule. Narrowing extends the *use* of ++ without altering its *definition* in a remarkable way. Even if all or part of either argument of ++ is unknown, (i.e., is an uninstantiated variable), narrowing keeps computing. In principle, this is not difficult—a value is assigned to the unknown parts of an argument—but the technical details that make this approach practical (sound, complete, and as efficient as the best functional computation when the arguments are fully known) are non-trivial and are the central subject of this paper. For the time being, we only want to show the advantages provided by narrowing in both declarative programming, in general, and functional logic languages, in particular. We begin with the latter.

A major obstacle in the integration of functional and logic programming is the evaluation of functional expressions containing logic uninstantiated variables. Narrowing, for its very nature, is obviously an elegant solution to this problem. An alternative solution is to *residuate*, i.e., to delay the evaluation of expressions containing uninstantiated variables until they are more instantiated, but there is no guarantee that these expressions will later become instantiated enough to be evaluated.

Economy of code, both textual and conceptual, is one of the advantages of narrowing. For example, any abstraction of a type *List* which defines the concatenation operation is likely to define several other functions such as *split* (a function that given a list L returns two lists X and Y that concatenated together produce L), *isPrefix* (a function that given two lists X and L tells whether there exists a list Y such that X concatenated with Y produces L), and many others. Narrowing makes (the definitions of) *split*, *isPrefix*, and many other functions superfluous. The advantage for the programmer is not only the time and effort saved during software development (because many function definitions can be omitted), but also, and perhaps even more so, during software maintenance.

In any abstraction there are groups of functions, such as ++, *split*, and *isPrefix*, that are intimately related, but neither the groups nor the relationships between the functions in a group are explicit in the code. When a function in a group changes due to maintenance, other functions in the same groups might have to change accordingly. Without a good understanding of the code, the programmer cannot know which functions should change or how. In this situation, the potential of introducing errors that are difficult to find and correct is high. Since narrowing allows us to avoid defining many related functions, these maintenance errors can no longer occur.

We demonstrate how the definition of related functions becomes superfluous by continuing our example. The key is the ability to solve equations. The details of this activity will be a central issue of this paper. For the time being, our discussion is informal and intuitive. The definition of ++ is used to split a list L into two lists X and Y by solving the equation $X \mathbin{++} Y \approx L$. For instance, solving the equation $X \mathbin{++} Y \approx [a, b]$ yields the three solutions $\{X \mapsto [], Y \mapsto [a, b]\}$, $\{X \mapsto [a], Y \mapsto [b]\}$, and $\{X \mapsto [a, b], Y \mapsto []\}$. Similarly, we compute a prefix P of a list L (or check whether P is a prefix of L, if P is instantiated) by solving the equation $P \mathbin{++} \_ \approx L$ ($\_$ denotes an anonymous variable). Likewise, the last element E of a list L is computed by solving the equation $\_ \mathbin{++} [E] \approx L$.

The ability to solve equations over user-defined types is an essential feature in application programs to describe problems in a declarative and readable manner. For instance, a classic solution to the 8-queens problem checks whether two queens can capture each other. Within a program, this task is translated into selecting two distinct elements in a list L and checking whether they satisfy a given property p. The selection of two distinct elements in a list does not require defining any additional function beside ++. The entire task is simply coded as

$$\_ \mathbin{++} [E] \mathbin{++} \_ \mathbin{++} [F] \mathbin{++} \_ \approx L \;\wedge\; p(E, F)$$

In order to solve equations between expressions containing defined functions, most proposals for the integration of functional and logic programming languages are based on narrowing, e.g., [10, 22, 24, 29, 55, 65]. Narrowing, introduced in automated theorem proving [67], is a relation over terms induced by a term rewriting system. For a given term rewriting system $\mathcal{R}$, narrowing is used to *solve* equations by computing unifiers with respect to the equational theory defined by $\mathcal{R}$ [21].

Informally, narrowing unifies a term with the left-hand side of a rewrite rule and fires the rule on the instantiated term.

Example 2 Consider the following rewrite rules defining the operations "less than or equal to" and addition for natural numbers represented by terms built with 0 and s:

$$\begin{array}{rcll} 0 \leq X & \to & \mathit{true} & \mathrm{R}_1 \\ s(X) \leq 0 & \to & \mathit{false} & \mathrm{R}_2 \\ s(X) \leq s(Y) & \to & X \leq Y & \mathrm{R}_3 \end{array} \qquad \begin{array}{rcll} 0 + X & \to & X & \mathrm{R}_4 \\ s(X) + Y & \to & s(X+Y) & \mathrm{R}_5 \end{array}$$

The rules defining "$\leq$" will be used in following examples. To narrow the equation $Z + s(0) \approx s(s(0))$, rule R_5 is applied by instantiating Z to $s(X)$. To narrow the resulting equation, $s(X + s(0)) \approx s(s(0))$, R_4 is applied by instantiating X to 0. The resulting equation, $s(s(0)) \approx s(s(0))$, is trivially true. Thus, $\{Z \mapsto s(0)\}$ is the equation's solution.

To apply the general idea of narrowing to integrated functional logic languages, a *functional logic program* is considered as a set of rewrite rules (with some additional restrictions explained later). A computation is initiated by solving an equation, $t \approx t'$, which possibly contains variables. This includes goal solving, as in logic programming, as well as evaluating expressions as in functional programming. An expression e can be evaluated by solving the equation $X \approx e$ so that X is instantiated to the result of evaluating e.

An important aspect in the design of functional logic languages is the definition of an appropriate strategy to solve equations. Such a strategy should be *sound* (i.e., only correct solutions are computed) and *complete* (i.e., all solutions or more general representatives of all solutions are computed). The narrowing relation can be used to define such a strategy, but a brute-force approach to finding all the solutions of an equation would attempt to unify *each* rule with *each* non-variable subterm of the given equation in every narrowing step. The resulting search space would be huge, even for small rewrite programs. Therefore, many narrowing strategies for limiting the size of the search space have been proposed, e.g., basic [42], innermost [22], outermost [17], standard [14, 43, 52, 73], lazy [23, 28, 54, 55, 65], or narrowing with redundancy tests [11]. Each strategy demands certain conditions of the rewrite relation to ensure the completeness of narrowing (the ability to compute all the solutions of an equation.)

Example 3 Consider the rewrite rules of Example 2. Although the equation $(X+X)+X \approx 0$ has only one solution, *eager* narrowing strategies that evaluate argument terms first have an infinite search space for this equation by always applying rule R_5 to the innermost term headed by $+$. This infinite derivation can be avoided by a *lazy* narrowing strategy which evaluates a term only if it is demanded. The right definition of "demanded" is a subtle point since the "demandedness" of a term may depend on the instantiation of other variables. For instance, consider the term $X \leq Y + Y$. The evaluation of the second argument $Y + Y$ is not demanded if X is instantiated to 0, but it is demanded if X is instantiated to $s(\cdots)$. A lazy narrowing strategy where the notion of "demanded" is independent of variable instantiations is defined in [55]. Thus, this strategy may perform superfluous steps and computes answers that are too specialized. This fact motivated various improvements, (e.g., [28, 49, 54, 56]) but none of them could show that the performed computation steps are really necessary and cannot be avoided.

Our contribution is a strategy that, for *inductively sequential* systems [2], preserves the completeness of narrowing and performs only steps that are "unavoidable" for solving equations. This characterization leads to the optimality of our strategy with respect to the number of "distinct" steps of a derivation. Advantages of our strategy over existing ones include: the large class of rewrite systems to which it is applicable, both the optimality of the derivations and the incomparability of the unifiers it computes, and the ease of its implementation.

The notion of an unavoidable step is well-known for rewriting. *Orthogonal* systems have the property that in every term t not in normal form, there exists a redex called *needed* that must "eventually" be reduced to compute the normal form of t [41, 48, 60]. Furthermore, repeated rewriting of needed redexes in a term suffices to compute its normal form, if it exists. Loosely speaking, only needed redexes really matter for rewriting in orthogonal systems. We extend this fact to narrowing in inductively sequential systems—a subclass of the orthogonal systems.

We also present a second optimality result (which cannot be stated for rewriting derivation) concerned with the substitution computed by a narrowing derivation. Every derivation of a same equation computes a set of substitutions. Different derivations compute disjoint sets of substitutions. This property nicely complements the neededness of a step in that the derivations computed by the strategy are needed in their entirety as well. Any solution computed by a derivation is not computed by any other derivation; hence, every derivation leading to a solution is needed as well as any step of the derivation.

Restricting our discussion to inductively sequential systems is not a limitation for the use of narrowing in programming languages. In fact, inductively sequential systems model the first-order functional component of programming languages, such as *ML* and *Haskell*, that establish priorities among rules by textual precedence or specificity [45]. Computing a needed redex in a term is an unsolvable problem. *Strongly sequential* systems are a large class for which the problem has an efficient solution. Inductively sequential systems are constructor-based and strongly sequential [2]. It has been shown both that inductively sequential systems are the constructor-based subclass of strongly sequential systems [34] and that a meaningful notion of needed redex can be formulated for *overlapping* inductively sequential systems [4].

After some preliminaries in Section 2, we present our strategy in Section 3. We formulate the soundness and completeness results in Section 4. We address our strategy's optimality in Section 5. Section 6 outlines several recent extensions of our strategy. We compare related work in Section 7. Our conclusion is in Section 8.

2 Preliminaries

We recall some key notions and notations about rewriting. We are consistent with the conventions of [15, 47]. First of all, we fix the notations for terms.

Definition 1 A many-sorted *signature* Σ is a pair (S, Ω) where S is a set of *sorts* and Ω is a family of *operation* sets of the form $\Omega = (\Omega_{w,s} | w \in S^*, s \in S)$. Let $\mathcal{X} = (\mathcal{X}_s | s \in S)$ be an S-sorted, countably infinite set of *variables*. Then the set $\mathcal{T}(\Sigma, \mathcal{X})_s$ of *terms* of sort s built from Σ and $\mathcal{X}$ is the smallest set containing $\mathcal{X}_s$ such that $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{X})_s$ whenever $f \in \Omega_{(s_1,\ldots,s_n),s}$ and $t_i \in \mathcal{T}(\Sigma, \mathcal{X})_{s_i}$. If $f \in \Omega_{\epsilon,s}$, we write f instead of $f()$. $\mathcal{T}(\Sigma, \mathcal{X})$ denotes the set of all terms. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term t is called *ground term* if $\mathcal{V}ar(t) = \emptyset$. A term is called *linear* if it does not contain multiple occurrences of one variable. In the following, Σ stands for a many-sorted signature.

In practice, most equational logic programs are *constructor-based*; symbols, called *constructors*, that construct data terms are distinguished from those, called *defined functions* or *operations*, that operate on data terms (see, for instance, the Equational Interpreter [61] and the functional logic languages *ALF* [29], *BABEL* [55], *K-LEAF* [23], *LPG* [10], *SLOG* [22]). Hence we define:

Definition 2 The set of operations Ω of a signature Σ is partitioned into two disjoint sets $\mathcal{C}$ and $\mathcal{D}$. $\mathcal{C}$ is the set of *constructors* and $\mathcal{D}$ is the set of *defined operations*. The terms in $\mathcal{T}(\mathcal{C},\mathcal{X})$ are called *constructor terms*. A term $f(t_1,\ldots,t_n)$ $(n \geq 0)$ is called a *pattern* if $f \in \mathcal{D}$ and $t_1,\ldots,t_n$ are constructor terms. A term $f(t_1,\ldots,t_n)$ $(n \geq 0)$ is called *operation-rooted term* (respectively *constructor-rooted term*) if $f \in \mathcal{D}$ (respectively $f \in \mathcal{C}$). A *constructor-based term rewriting system* $\mathcal{R}$ is a set of *rewrite rules*, $l \to r$, such that l and r have the same sort, l is a pattern, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$.

In the rest of this paper we assume that $\mathcal{R}$ is a *constructor-based term rewriting system*. Substitutions are an essential concept to define the notions of rewriting and narrowing.

Definition 3 A *substitution* is a mapping $\sigma\colon \mathcal{X} \to \mathcal{T}(\Sigma,\mathcal{X})$ with $\sigma(x) \in \mathcal{T}(\Sigma,\mathcal{X})_s$ for all variables $x \in X_s$ such that its *domain* $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is finite. We frequently identify a substitution σ with the set $\{x \mapsto \sigma(x) \mid x \in \mathcal{D}om(\sigma)\}$. The *image* of a substitution σ is the set of variables $\mathcal{I}m(\sigma) = \{y \in \mathcal{V}ar(\sigma(x)) \mid x \in \mathcal{D}om(\sigma)\}$. Substitutions are extended to morphisms on $\mathcal{T}(\Sigma,\mathcal{X})$ by $\sigma(f(t_1,\ldots,t_n)) = f(\sigma(t_1),\ldots,\sigma(t_n))$ for every term $f(t_1,\ldots,t_n)$. A substitution σ is called (*ground*) *constructor substitution* if $\sigma(x)$ is a (ground) constructor term for all $x \in \mathcal{D}om(\sigma)$. The *composition of two substitutions* σ and τ is defined by $(\sigma \circ \tau)(x) = \sigma(\tau(x))$ for all $x \in \mathcal{X}$. The *restriction* $\sigma_{|V}$ of a substitution σ to a set V of variables is defined by $\sigma_{|V}(x) = \sigma(x)$ if $x \in V$ and $\sigma_{|V}(x) = x$ if $x \notin V$. A substitution σ is *more general* than σ', denoted by $\sigma \leq \sigma'$, if there is a substitution τ with $\sigma' = \tau \circ \sigma$. If V is a set of variables, we write $\sigma = \sigma'[V]$ iff $\sigma_{|V} = \sigma'_{|V}$, and we write $\sigma \leq \sigma'[V]$ iff there is a substitution τ with $\sigma' = \tau \circ \sigma[V]$. A substitution σ is *idempotent* iff $\sigma \circ \sigma = \sigma$.

A term t' is an *instance* of t if there is a substitution σ with $t' = \sigma(t)$. In this case we write $t \leq t'$. A term t' is a *variant* of t if $t \leq t'$ and $t' \leq t$.

A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. A unifier σ is called *most general* (*mgu*) if $\sigma \leq \sigma'$ for every other unifier σ'. Most general unifiers are unique up to variable renaming. By introducing a total ordering on variables we can uniquely choose *the* most general unifier of two terms. Hence we denote by $mgu(s,t)$ the most general unifier of s and t.

All the unifiers considered in this paper will be computed from a term and a linear pattern whose set of variables are disjoint. Under these conditions it is easy to verify that, restricted to the variables of these terms, only idempotent substitutions are computed. Therefore, we implicitly assume in our proofs that a unifier is an idempotent substitution and that any variable in the domain of a unifier is already contained in one of the terms being unified.

We now introduce positions, which are essential to define the notions of rewriting and narrowing.

Definition 4 An *occurrence* or *position* is a sequence of positive integers identifying a subterm in a term. For every term t, the empty sequence denoted by Λ, identifies t itself. For every term of the form $f(t_1,\ldots,t_k)$, the sequence $i \cdot p$, where i is a positive integer not greater than k and p is a position, identifies the subterm of t_i at p. The subterm of t at p is denoted by $t|_p$ and the result of *replacing* $t|_p$ with s in t is denoted by $t[s]_p$. If p and q are positions, we write $p \leq q$ if p is *above* or is a *prefix* of q, and we write $p \parallel q$ if the positions are *disjoint* (see [15] for details). The expression $p \cdot q$ denotes the position resulting from the concatenation of the positions p and q, i.e., we overload the symbol "$\cdot$."

We are now ready to define rewriting.

Definition 5 A *reduction step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there exist a position p, a rewrite rule $R = l \to r$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$.

In this case we say t is *rewritten* (at position p) *to* s and $t|_p$ is a *redex* of t. We will omit the subscripts p and R if they are clear from the context. A redex $t|_p$ of t is an *outermost redex* if there is no redex $t|_q$ of t with $q < p$. $\overset{*}{\rightarrow}$ denotes the transitive and reflexive closure of $\rightarrow$. $\overset{*}{\leftrightarrow}$ denotes the transitive, reflexive and symmetric closure of $\rightarrow$. A term t is *reducible to* a term s if $t \overset{*}{\rightarrow} s$. A term t is called *irreducible* or in *normal form* if there is no term s with $t \rightarrow s$. A term s is a *normal form of* t if t is reducible to the irreducible term s.

Rewriting is computing, i.e., the *value* of a functional expression is its normal form obtained by rewriting. Functional logic programs compute with partial information, i.e., a functional expression may contain logic variables. The goal is to compute values for these variables such that the expression is evaluable to a particular normal form, e.g., a constructor term [23, 55]. This is done by narrowing.

Definition 6 A term t is *narrowable* to a term s if there exist a non-variable position p in t (i.e., $t|_p \notin \mathcal{X}$), a variant $l \rightarrow r$ of a rewrite rule in $\mathcal{R}$ with $\mathcal{V}ar(t) \cap \mathcal{V}ar(l \rightarrow r) = \emptyset$ and a unifier σ of $t|_p$ and l such that $s = \sigma(t[r]_p)$. In this case we write $t \rightsquigarrow_{p,l\rightarrow r,\sigma} s$. If σ is a most general unifier of $t|_p$ and l, the narrowing step is called *most general.* We write $t_0 \overset{*}{\rightsquigarrow}_\sigma t_n$ if there is a narrowing derivation $t_0 \rightsquigarrow_{p_1,R_1,\sigma_1} t_1 \rightsquigarrow_{p_2,R_2,\sigma_2} \cdots \rightsquigarrow_{p_n,R_n,\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_2 \circ \sigma_1$.

Since the instantiation of the variables in the rule $l \rightarrow r$ by σ is not relevant for the computed result of a narrowing derivation, we will omit this part of σ in the example derivations in this paper.

Example 4 Referring to Example 2,

$$A + B \rightsquigarrow_{\Lambda,\mathrm{R}_5,\{A\mapsto s(0),B\mapsto 0\}} s(0+0)$$

and

$$A + B \rightsquigarrow_{\Lambda,\mathrm{R}_5,\{A\mapsto s(X)\}} s(X+B)$$

are narrowing steps of $A + B$, but only the latter is a most general narrowing step.

Padawitz [62] also distinguishes between narrowing and most general narrowing but, in most papers, narrowing is intended as most general narrowing (e.g., [42]). Most general narrowing has the advantage that most general unifiers are uniquely computable, whereas there exist many distinct unifiers. Dropping the requirement that unifiers be most general is crucial to the definition of a needed narrowing step since these steps may be impossible with most general unifiers.

Narrowing solves equations, i.e., computes values for the variables in an equation such that the equation becomes true, where an *equation* is a pair $t \approx t'$ of terms of the same sort. In a constructor-based setting, it is reasonable to consider only ground constructor terms as values and to require that an equation holds if both sides have the same value (see also [23] for a more detailed discussion on this topic). Since we do not require terminating term rewriting systems, normal forms or values do not exist for each functional expression. Hence, we define the validity of an equation as a strict equality on terms in the spirit of functional logic languages with a lazy operational semantics such as *K-LEAF* [23] and *BABEL* [55]; an equation is satisfied if both sides are equivalent to a same ground constructor term. This is formally expressed by the following definition (since we consider in this paper only confluent rewrite systems).

Definition 7 An *equation* is a pair $t \approx t'$ of terms of the same sort. A substitution σ is a *solution* for an equation $t \approx t'$ iff $\sigma(t)$ and $\sigma(t')$ are reducible to a same ground constructor term.

Our definition of solution is weaker than convertibility, i.e., $\sigma(t) \overset{*}{\leftrightarrow} \sigma(t')$. This is due to the fact that we are discussing constructor-based, not necessarily terminating rewrite systems.

Equations can also be interpreted as terms by defining the symbol $\approx$ as a binary operation symbol, more precisely, one operation symbol for each sort. Therefore, all notions for terms, such as substitution, rewriting, narrowing etc., will also be used for equations. The semantics of $\approx$ are 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.

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

These are the *equality rules* of a signature (this encoding of strict equality as rewrite rules is analogous to the encoding equality by the rule $x \approx x \to \mathit{true}$ in completion-based approaches to equation solving [16]). It is easy to see that the orthogonality status of a rewrite system (see below) is not changed by these rules. The same holds true for the inductive sequentiality, which will be defined shortly. With these rules, a solution of an equation is computed by narrowing it to *true*—an approach also taken in *K-LEAF* [23] and *BABEL* [55]. The equivalence between the reducibility to a same ground constructor term and the reducibility to *true* using the equality rules is addressed by Proposition 1.

We also require orthogonality, which ensures the good-behavior of computations.

Definition 8 A term rewriting system $\mathcal{R}$ is *orthogonal* if for each rule $l \to r \in \mathcal{R}$ the left-hand side l is linear (*left-linearity*) and for each non-variable subterm $l|_p$ of l there exists no rule $l' \to r' \in \mathcal{R}$ such that $l|_p$ and l' unify (*non-overlapping*) (where $l' \to r'$ is not a variant of $l \to r$ in case of $p = \Lambda$).

Our strategy extends to narrowing the rewriting notion of *need*. The idea, for rewriting, is to reduce in a term only certain redexes which *must* be reduced to compute the normal form of t. In orthogonal term rewriting systems, every term not in normal form has a redex that must be reduced to compute the term's normal form. The following definition [41] formalizes this idea.

Definition 9 Let $A = t \to_{u,l\to r} t'$ be a reduction step of some term t into t' at position u with rule $l \to r$. The set of *descendants* (or *residuals*) of a position v by A, denoted $v \setminus A$, is

$$v \setminus A = \begin{cases} \emptyset & \text{if } v = u \cdot p \text{ and } l|_p \text{ is not a variable,} \\ \{v\} & \text{if } u \not\leq v, \\ \{u \cdot p' \cdot q \ \text{ such that } \ r|_{p'} = x\} & \text{if } v = u \cdot p \cdot q \text{ and } l|_p = x, \text{ where } x \text{ is a variable.} \end{cases}$$

The set of *descendants* of a position v by a reduction sequence B is defined by induction as follows

$$v \setminus B = \begin{cases} \{v\} & \text{if } B \text{ is the null derivation,} \\ \displaystyle\bigcup_{w \in v \setminus B'} w \setminus B'' & \text{if } B = B'B'', \text{ where } B' \text{ is the initial step of } B. \end{cases}$$

A position u of a term t is called *needed* iff in every reduction sequence of t to a normal form a descendant of $t|_u$ is rewritten at its root.

A position uniquely identifies a subterm of a term. The notion of *descendant* for terms stems directly from the corresponding notion for positions.

A more intuitive definition of descendant of a position or term is proposed in [48]. Let $t \overset{*}{\to} t'$ be a reduction sequence and s a subterm of t. The descendants of s in t' are computed as follows: underline the root of s and perform the reduction sequence $t \overset{*}{\to} t'$. Then, every subterm of t' with an underlined root is a *descendant* of s.

Example 5 Consider the operation that doubles its argument by means of an addition. The rules of addition are in Example 2.

$$double(X) \;\to\; X + X \qquad \mathrm{R}_6$$

In the following reduction of $double(0 + 0)$ we show, by means of underlining, the descendants of $0 + 0$.

$$double(0\underline{\;+\;}0) \to_{\Lambda, \mathrm{R}_6} (0\underline{\;+\;}0) + (0\underline{\;+\;}0)$$

The set of descendants of position 1 by the above reduction is $\{1, 2\}$.

3 Outermost-needed narrowing

An efficient narrowing strategy must limit the search space. No suitable rule can be ignored, but some positions in a term may be neglected without losing completeness. For instance, Hullot [42] has introduced *basic narrowing*, where narrowing is not applied at positions introduced by substitutions. Fribourg [22] has proposed *innermost narrowing*, where narrowing is applied only to a pattern. Hölldobler [39] has combined innermost and basic narrowing. Narrowing only at *outermost* positions is complete only if the rewrite system satisfies strong restrictions such as non-unifiability of subterms of the left-hand sides of rewrite rules [17]. *Lazy narrowing* [23, 54, 65], akin to lazy evaluation in functional languages, attempts to avoid unnecessary evaluations of expressions. A lazy narrowing step is applied at outermost positions with the exception that inner arguments of a function are evaluated, by narrowing them to their head normal forms, if their values are required for an outermost narrowing step. Unfortunately, the property "required" depends on the rules tried in following steps, but looking-ahead is not a viable option.

We want to perform only narrowing steps that are necessary for computing solutions. Naively, one could say that a narrowing step $t \rightsquigarrow_{p, l \to r, \sigma} t'$ is *needed* iff p is a position of t, σ is the most general unifier of $t|_p$ and l, and $\sigma(t|_p)$ is a needed redex. Unfortunately, a substantial complication arises from this simple approach. If t' is a normal form, the step is trivially needed. However, some instantiation performed later in the derivation could "undo" this need.

Example 6 Referring to Example 2, consider the term $t = X \leq Y + Z$. According to the naive approach, the following narrowing step of t at position 2

$$X \leq Y + Z \rightsquigarrow_{2, \mathrm{R}_4, \{Y \mapsto 0\}} X \leq Z$$

would be needed since $X \leq Z$ is a normal form. This step is indeed necessary to solve the inequality if $s(x)$ for some term x is eventually substituted for X, although this claim may not be obvious without the results presented in this paper. However, the same step becomes unnecessary if 0 is substituted for X. The following derivation computes a more general solution of the inequation without ever narrowing any descendant of t at 2.

$$X \leq Y + Z \rightsquigarrow_{\Lambda, \mathrm{R}_1, \{X \mapsto 0\}} true$$

Thus, in our definition, we impose a condition strong enough to ensure the necessity of a narrowing step, no matter which unifiers might be used later in the derivation.

Definition 10 A narrowing step $t \rightsquigarrow_{p, R, \sigma} t'$ is called *needed* or *outermost-needed* iff, for every $\eta \geq \sigma$, p is the position of a needed or outermost-needed redex of $\eta(t)$, respectively. A narrowing derivation is called *needed* or *outermost-needed* iff every step of the derivation is needed or outermost-needed, respectively.

Our definition adds, with respect to rewriting, a new dimension to the difficulty of computing needed narrowing steps. We must take into account any instantiation of a term in addition to any derivation to normal form. Luckily, as for rewriting, the problem has an efficient solution in inductively sequential systems. We forgo the requirement that the unifier of a narrowing step be most general. The instantiation that we demand in addition to that for the most general unification ensures the need of the position irrespective of future unifiers. It turns out that this extra instantiation would eventually be performed later in the derivation. Thus, we are only "anticipating" it and the completeness of narrowing is preserved. This approach, however, complicates the notion of narrowing strategy.

According to [17, 62], a narrowing strategy is a function from terms into non-variable positions in these terms so that exactly one position is selected for the next narrowing step. Unfortunately, this notion of narrowing strategy is inadequate for narrowing with arbitrary unifiers which, as Example 6 shows, are essential to capture the need of a narrowing step.

Definition 11 A *narrowing strategy* is a function from terms into sets of triples. If $\mathcal{S}$ is a narrowing strategy, t is a term, and $(p, l \to r, \sigma) \in \mathcal{S}(t)$, then p is a position of t, $l \to r$ is a rewrite rule, and σ a substitution such that $t \rightsquigarrow_{p, l \to r, \sigma} \sigma(t[r]_p)$ is a narrowing step.

We now define a class of rewrite systems for which there exists an efficiently computable needed narrowing strategy. Systems in this class have the property that the rules defining any operation can be organized in a hierarchical structure called definitional tree [2], which is used to implement needed rewriting. This paper generalizes that result to narrowing.

The symbols *branch* and *leaf*, used in the next definition, are uninterpreted functions used to classify the nodes of the tree. A definitional tree can be seen as a partially ordered set of patterns with some additional constraints.

Definition 12 $\mathcal{T}$ is a *partial definitional tree*, or *pdt*, with pattern π iff one of the following cases holds:

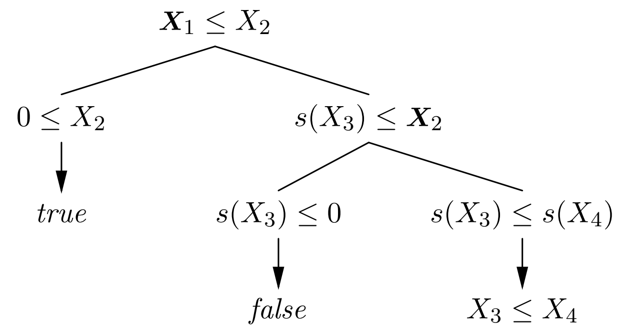
$\mathcal{T} = \mathit{branch}(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, where π is a pattern, o is the occurrence of a variable of π, the sort of $\pi|_o$ has constructors $c_1, \ldots, c_k$, for some $k > 0$, and for all i in $\{1, \ldots, k\}$, $\mathcal{T}_i$ is a *pdt* with pattern $\pi[c_i(X_1, \ldots, X_n)]_o$, where n is the arity of c_i and $X_1, \ldots, X_n$ are new distinct variables.

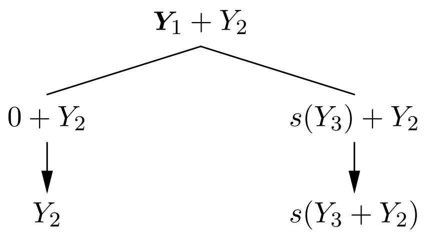
$\mathcal{T} = \mathit{leaf}(\pi)$, where π is a pattern.

We denote by $\mathcal{P}(\Sigma)$ the set of *pdt*s over the signature Σ. Let $\mathcal{R}$ be a rewrite system. $\mathcal{T}$ is a *definitional tree* of an operation f iff $\mathcal{T}$ is a *pdt* whose pattern argument is $f(X_1, \ldots, X_n)$, where n is the arity of f and $X_1, \ldots, X_n$ are new distinct variables, and for every rule $l \to r$ of $\mathcal{R}$ with $l = f(t_1, \ldots, t_n)$, there exists a leaf $\mathit{leaf}(\pi)$ of $\mathcal{T}$ such that l is a variant of π, and we say that the node $\mathit{leaf}(\pi)$ *represents* the rule $l \to r$. We call *minimal* a definitional tree $\mathcal{T}$ of an operation f iff below any *branch* node of $\mathcal{T}$ there is a *leaf* representing a rule defining f.

We call *inductively sequential* an operation, f, of a rewrite system, $\mathcal{R}$, iff there exists a definitional tree $\mathcal{T}$ of f such that each *leaf* node of $\mathcal{T}$ represents at most one rule of $\mathcal{R}$. We call *inductively sequential* a rewrite system $\mathcal{R}$ iff any operation of $\mathcal{R}$ is inductively sequential.

Example 7 We show pictorial representations of definitional trees of the operations defined in Example 2. A branch node of the picture shows the pattern of a corresponding node of the definitional tree. Every leaf node represents a rule. We show the right side of each such rule below the pattern of the leaf which is connected by an arrow. The occurrence argument of a branch node is shown by emboldening the corresponding subterm in the pattern argument.





Distinguishing whether or not a *leaf* node of a definitional tree of some operation f represents a rule defining f is sometimes important in our treatment. We write *exempt*(π) instead of *leaf*(π) to point out that *leaf*(π) is a *pdt* that does not represent any rule. Likewise, we abbreviate with *rule*$(\pi, \sigma(l) \to \sigma(r))$ the fact that *leaf*(π) is a *pdt* representing some rule $l \to r$ of the considered rewrite system, where σ is the renaming substitution such that $\sigma(l) = \pi$. The patterns of a definitional tree are a finite set partially ordered by the subsumption preordering. The set of the patterns occurring within the leaves of a definitional tree is complete w.r.t. the set of constructors in the sense of [40]. Consequently, the defined functions of an inductively sequential term rewriting system are completely defined over their application domains [26, 69] (i.e., any ground term has a constructor term as a normal form) if the considered term rewriting system is terminating and the possible definitional trees do not contain *exempt* nodes.

We now give an informal account of our strategy. Let $t = f(t_1, \ldots, t_k)$ be a term to narrow. We unify t with some maximal element of the set of patterns of a definitional tree of f. Let π denote such a pattern, τ the most general unifier of t and π, and $\mathcal{T}$ the *pdt* in which π occurs. If $\mathcal{T}$ is a *rule pdt*, then we narrow $\tau(t)$ at the root with the rule represented by $\mathcal{T}$. If $\mathcal{T}$ is an *exempt pdt*, then $\tau(t)$ cannot be narrowed to a constructor-rooted term. If $\mathcal{T}$ is a *branch pdt*, then we recur on $\tau(t|_o)$, where o is the occurrence contained in $\mathcal{T}$ and τ is the *anticipated* substitution. The result of the recursive invocation is suitably composed with τ and o. The details of this composition are in the formal definition presented below.

We derive our outermost-needed strategy from a mapping, λ, that implements the above computation. λ takes an operation-rooted term, t, and a definitional tree, $\mathcal{T}$, of the root of t, and non-deterministically returns a triple, (p, R, σ), where p is a position of t, R is either a rule $l \to r$ of $\mathcal{R}$ or the distinguished symbol "?," and σ is a substitution. If $R = l \to r$, then our strategy performs the narrowing step $t \leadsto_{p, l \to r, \sigma} \sigma(t[r]_p)$. If $R = ?$, then our strategy gives up, since it is impossible to narrow t to a constructor-rooted term.

In the following, $pattern(\mathcal{T})$ denotes the pattern argument of $\mathcal{T}$, and $\prec$ denotes the Noetherian ordering on $\mathcal{T}(\Sigma, \mathcal{X}) \times \mathcal{P}(\Sigma)$ defined by: $(t_1, \mathcal{T}_1) \prec (t_2, \mathcal{T}_2)$ if and only if either: (*i*) t_1 has fewer occurrences of defined operation symbols than t_2 or (*ii*) $t_1 = t_2$ and $\mathcal{T}_1$ is a proper subtree of $\mathcal{T}_2$.

Definition 13 The function λ takes two arguments: an operation-rooted term, t, and a *pdt*, $\mathcal{T}$, such that $pattern(\mathcal{T})$ and t unify. The function λ yields a set of triples of the form (p, R, σ), where p is a position of t, R is either a rewrite rule or the distinguished symbol "?," and σ is a unifier of $pattern(\mathcal{T})$ and t. Thus, let t be a term and $\mathcal{T}$ a *pdt* in the domain of λ. The function λ is defined by induction on $\prec$ as follows.

$$\lambda(t,\mathcal{T}) \ni \begin{cases} (\Lambda, R, mgu(t,\pi)) & \text{if } \mathcal{T} = rule(\pi, R); \\ (\Lambda, ?, mgu(t,\pi)) & \text{if } \mathcal{T} = exempt(\pi); \\ (p, R, \sigma) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad t \text{ and } pattern(\mathcal{T}_i) \text{ unify, for some } i, \text{ and} \\ & \quad (p, R, \sigma) \in \lambda(t, \mathcal{T}_i); \\ (o \cdot p, R, \sigma \circ \tau) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad t \text{ and } pattern(\mathcal{T}_i) \text{ do not unify, for any } i, \\ & \quad \tau = mgu(t, \pi), \\ & \quad \mathcal{T}' \text{ is a definitional tree of the root of } \tau(t|_o), \text{ and} \\ & \quad (p, R, \sigma) \in \lambda(\tau(t|_o), \mathcal{T}'). \end{cases}$$

The function λ is trivially well-defined in the third case. By the definition of *pdt*, there exists a proper sub*pdt* $\mathcal{T}_i$ of $\mathcal{T}$ such that $pattern(\mathcal{T}_i)$ and t unify if $t|_o$ is constructor-rooted or a variable. Similarly, λ is well-defined in the fourth case since this case can only occur if $t|_o$ is operation-rooted. In this case, $\tau_{|\mathcal{V}ar(t)}$ is a constructor substitution since π is a linear pattern. Since t is operation-rooted and $o \neq \Lambda$, $\tau(t|_o)$ has fewer occurrences of defined operation symbols than t. Since $t|_o$ is operation-rooted, so is $\tau(t|_o)$. By the definition of *pdt*, $pattern(\mathcal{T}') \leq \tau(t|_o)$, i.e., $pattern(\mathcal{T}')$ and $\tau(t|_o)$ unify. This implies that λ is well-defined in this case as well.

As in proof procedures for logic programming, we have to apply *variants* of the rewrite rules *with fresh variables* to the current term. Therefore, we assume in the following that the definitional trees contain new variables if they are used in a narrowing step.

The computation of $\lambda(t,\mathcal{T})$ may entail a non-deterministic choice when $\mathcal{T}$ is a *branch pdt*—the integer i when $t|_o$ is a variable. The substitution τ, when $t|_o$ is operation-rooted, is the *anticipated* substitution guaranteeing the need of the computed position. It is pushed down in the recursive call to λ to ensure the consistency of the computation when t is non-linear. The anticipated substitution is neglected when $t|_o$ is not operation-rooted since the pattern in $\mathcal{T}_i$ is an instance of π. Hence, σ extends the anticipated substitution.

Example 8 We trace the computation of λ for the initial step of a derivation of $X \leq Y + Z$ as discussed in Example 6.

$$\begin{array}{l} \lambda(X \leq Y + Z, branch(X_1 \leq X_2, 1, \ldots)) \\ \quad \lambda(X \leq Y + Z, branch(s(X_3) \leq X_2, 2, \ldots)) \\ \quad\quad \lambda(Y + Z, branch(Y_1 + Y_2, 1, \ldots)) \\ \quad\quad\quad \lambda(Y + Z, rule(0 + Y_2, \mathrm{R}_4)) \\ \quad\quad\quad (\Lambda, \mathrm{R}_4, \{Y \mapsto 0, Y_2 \mapsto Z\}) \\ \quad\quad (\Lambda, \mathrm{R}_4, \{Y \mapsto 0, Y_2 \mapsto Z\}) \\ \quad (2, \mathrm{R}_4, \{X \mapsto s(X_3), X_2 \mapsto 0 + Z, Y \mapsto 0, Y_2 \mapsto Z\}) \\ (2, \mathrm{R}_4, \{X \mapsto s(X_3), X_2 \mapsto 0 + Z, Y \mapsto 0, Y_2 \mapsto Z\}) \end{array}$$

Note that we consider narrowing of operation-rooted terms. This limitation shortens our discussion and suffices for solving equations (see proof of Theorem 4). Extending our results to constructor-rooted terms is straightforward. To compute an outermost-needed narrowing step of a constructor-rooted term, it suffices to compute an outermost-needed narrowing step of any of its maximal operation-rooted subterms.

We prove two simple technical lemmas concerning the mutual relationships between the patterns of the *pdt*s of a definitional tree:

Lemma 1 *Let $\mathcal{T}$ be a* pdt*, p and q two positions of $\mathcal{T}$, and π_p and π_q the patterns of the* pdt*s at the positions p and q of $\mathcal{T}$ respectively. If $p \leq q$, then $\pi_p \leq \pi_q$.*

Proof If $p \leq q$, then there exists a position, r, such that $q = p \cdot r$. The proof is by induction on the length of r. Base case: $r = \Lambda$ implies $p = q$ and consequently $\pi_p = \pi_q$. Induction step: $r = i \cdot r'$, for some positive integer i and position r'. Let $\mathcal{T}_p$ be the *pdt* of $\mathcal{T}$ at p. $\mathcal{T}_p = branch(\pi_p, o, \mathcal{T}_{p_1}, \ldots, \mathcal{T}_{p_k})$, for some position o, and *pdt*s $\mathcal{T}_{p_1}, \ldots, \mathcal{T}_{p_k}$, for some $k \geq i$. Let π_{p_i} be the pattern in $\mathcal{T}_{p_i}$. Since π_p is linear and π_{p_i} is obtained by instantiating with a constructor term the variable of π_p at o, $\pi_p < \pi_{p_i}$. By construction, π_q is equal to the pattern of the *pdt* of $\mathcal{T}_{p_i}$ at r'. By the induction hypothesis, $\pi_{p_i} \leq \pi_q$. By the transitivity of "$\leq$," $\pi_p < \pi_q$. □

Lemma 2 *The patterns of the* pdt*s at two disjoint positions of a* pdt *$\mathcal{T}$ do not unify.*

Proof The proof is by structural induction on the *pdt* $\mathcal{T}$.

Base case: $\mathcal{T} = leaf(\pi)$, for some pattern π.

There are no disjoint positions in $\mathcal{T}$ and the claim vacuously holds.

Induction step: $\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, for some pattern π, position o, and *pdt*s $\mathcal{T}_1, \ldots, \mathcal{T}_k$, for some $k > 0$.

Let p and q be two disjoint positions in $\mathcal{T}$. Both p and q differ from Λ; hence, there exist integers i and j in $\{1, \ldots, k\}$, and positions p' and q' such that $p = i \cdot p'$ and $q = j \cdot q'$. If $i = j$, then p' and q' are disjoint positions of $\mathcal{T}_i$. The patterns of $\mathcal{T}$ at p and q are equal to the patterns of $\mathcal{T}_i$ at p' and q', respectively. The latter do not unify by the induction hypothesis. If $i \neq j$ then, by Lemma 1, the patterns of $\mathcal{T}$ at p and q are instances of the root patterns of $\mathcal{T}_i$ and $\mathcal{T}_j$, respectively. The latter do not unify since they have a different symbol at position o; thus, the former do not unify either. □

Lemma 3 *If $\mathcal{R}$ is an inductively sequential rewrite system, then $\mathcal{R}$ is orthogonal.*

Proof Let f be any operation of $\mathcal{R}$. By definition of inductive sequentiality, there exists a definitional tree of f, $\mathcal{T}$, such that every rule defining f is represented by one leaf of $\mathcal{T}$. Thus, the left-hand sides of the rules defining f are linear since all patterns of a definitional tree are linear by definition. They are also non-overlapping by Lemma 2. Finally, since the left-hand sides are patterns, the rules of different operations do not overlap. Hence, $\mathcal{R}$ is orthogonal. □

We are interested only in narrowing derivations that end in a constructor term. Our key result is that if λ, on input of a term t, computes a position p and a substitution σ and η extends σ, then $\eta(t)$ must "eventually" be narrowed at p to obtain a constructor term. "Eventually" is formalized by the notion of *descendant* which, initially proposed for rewriting [41], is extended to narrowing simply by replacing $\rightarrow_{u,l \rightarrow r}$ with $\rightsquigarrow_{u,l \rightarrow r,\sigma}$ in Definition 9.

Theorem 1 *Let $\mathcal{R}$ be an inductively sequential rewrite system, t an operation-rooted term, and $\mathcal{T}$ a definitional tree of the root of t. Let $(p, R, \sigma) \in \lambda(t, \mathcal{T})$ and η extend σ, i.e., $\eta \geq \sigma$.*

1. *In any narrowing derivation of $\eta(t)$ to a constructor-rooted term a descendant of $\eta(t|_p)$ is narrowed to a constructor-rooted term.*
2. *If $R = l \rightarrow r$, then $t \rightsquigarrow_{p,R,\sigma} \sigma(t[r]_p)$ is an outermost-needed narrowing step.*
3. *If $R = ?$, then $\eta(t)$ cannot be narrowed to a constructor-rooted term.*

Proof We prove by Noetherian induction on $\prec$ the claim generalized by considering $\mathcal{T}$ a subtree of a definitional tree of the root of t such that *pattern*$(\mathcal{T})$ and t unify. We consider the cases of the definition of λ.

Base case: consider $(t, \mathcal{T})$ where t is an operation-rooted term and $\mathcal{T} = \textit{leaf}(\pi)$, for some pattern π. We consider the two subcases of the definition of λ for *leaf* nodes:

$\mathcal{T} = \textit{rule}(\pi, R')$, for some pattern π and rule R'.

In this case $(p, R, \sigma) = (\Lambda, R', \textit{mgu}(t, \pi))$. Since $\eta(t)$ is operation-rooted and is a descendant of itself, claim number 1 trivially holds. Let $R' = l \to r$, for some terms l and r. By the definition of a definitional tree, $\pi = l$; hence, $\sigma(l) = \sigma(t|_p)$. Thus, $t \rightsquigarrow_{p,R,\sigma} \sigma(t[r]_p)$ is a narrowing step. Since $\mathcal{R}$ is orthogonal (by Lemma 3), its redex schemes do not overlap; consequently, R keeps matching any descendant of $\eta(t)$ obtained by reductions strictly below Λ. Thus, $\eta(t)$ is a needed redex of itself and it is obviously outermost. Claim number 3 vacuously holds.

$\mathcal{T} = \textit{exempt}(\pi)$, for some pattern π.

In this case $(p, R, \sigma) = (\Lambda, ?, \textit{mgu}(t, \pi))$. Since $\eta \geq \sigma$, η also unifies π and t. We could extend $\mathcal{R}$ by changing the *exempt* node into a *rule* node in which the left-hand side of the rule is obviously π and the right-hand side is arbitrary. Thus, similar to the previous case, π would keep unifying with any descendant of $\eta(t)$ obtained by reductions strictly below the root. Thus, by Lemma 2, there exists no rule in $\mathcal{R}$ that would unify with $\eta(t)$. Thus, $\eta(t)$ cannot be narrowed to a constructor-rooted term which implies claim number 3 and, trivially, claim number 1. Claim number 2 vacuously holds.

Induction step: consider $(t, \mathcal{T})$ where t is an operation-rooted term and $\mathcal{T} = \textit{branch}(\pi, o, \mathcal{T}_1, \dots, \mathcal{T}_k)$, for some pattern π, position o, and *pdt*s $\mathcal{T}_1, \dots, \mathcal{T}_k$, for some $k > 0$. We consider the two subcases of the definition of λ for *branch* nodes.

$t|_o$ is either constructor-rooted or is a variable.

By the definition of *pdt*, there exists some i in $\{1, \dots, k\}$ such that *pattern*$(\mathcal{T}_i)$ and t unify. By the definition of λ, $\lambda(t, \mathcal{T}) = \lambda(t, \mathcal{T}_i)$. By the induction hypothesis, all the claims hold already for $\lambda(t, \mathcal{T}_i)$ and they are independent of $\mathcal{T}_i$.

$t|_o$ is operation-rooted.

By the hypothesis, π and t unify. Let $\tau = \textit{mgu}(t, \pi)$. Since $t|_o$ is operation-rooted, so is $\tau(t|_o)$. Let $\mathcal{T}'$ be a definitional tree of the root of $\tau(t|_o)$. Let $(p', R', \sigma') \in \lambda(\tau(t|_o), \mathcal{T}')$ such that $(p, R, \sigma) = (o \cdot p', R', \sigma' \circ \tau)$, where p' is a position of $t|_o$, R' is either a rule or "?," and σ' is a substitution. In this case $(t|_o)|_{p'} = t|_{o \cdot p'} = t|_p$. By Lemma 2, any rule whose left-hand side might unify with t is represented by a leaf of $\mathcal{T}_i$. If $l \to r$ is a rule represented by a leaf of $\mathcal{T}_i$ then, by Lemma 1, *pattern*$(\mathcal{T}_i) \leq l$. Thus, by the definition of definitional tree, l has a constructor symbol at position o. However, the case being considered assumes that t has an operation symbol at position o. Hence, in any narrowing derivation of $\eta(t)$ that includes a step at the root, a descendant of $\eta(t|_o)$ must be narrowed to a constructor-rooted term. Since t is operation-rooted, $\eta(t)$ is also operation-rooted. In any narrowing derivation of $\eta(t)$ to a constructor-rooted term a descendant of $\eta(t)$ is narrowed at the root and, consequently, a descendant of $\eta(t|_o)$ is narrowed to a constructor-rooted term. By the definition of λ, $\tau(t|_o)$ has fewer occurrences of operation symbols than t. Thus, by the induction hypothesis, for any $\eta' \geq \sigma'$, in any narrowing derivation of $\eta'(\tau(t|_o))$ to a constructor-rooted term a descendant of $\eta'(\tau(t|_p))$ is narrowed to a constructor-rooted term. Since $\eta \geq \sigma$, $\eta = \phi \circ \sigma$ for some substitution ϕ. Let $\eta' = \phi \circ \sigma' \geq \sigma'$, which implies $\eta'(\tau(t)) = \eta(t)$ since $\sigma = \sigma' \circ \tau$. Hence, in any narrowing derivation of $\eta(t|_o)$ to a constructor-rooted term, a descendant of $\eta(t|_p)$ is narrowed to a constructor-rooted term. Thus, claim number 1 holds by transitivity.

We consider the two cases for R'.

R' is a rule.

By the induction hypothesis, $\tau(t|_o) \rightsquigarrow_{p', R, \sigma'} \sigma'(\tau(t|_o)[r]_{p'})$ is an outermost-needed narrowing step; hence, $t \rightsquigarrow_{p, R, \sigma} \sigma(t[r]_p)$ is a narrowing step. The need of $\eta(t|_p)$ with respect to $\eta(t)$ is an immediate consequence of claim number 1. By the hypothesis, π and t unify and, by the definition of definitional tree, π is a pattern and o is a position of π. These conditions imply that there is only one operation symbol in $\sigma(t)$ above o: the root of $\sigma(t)$. In constructor-based systems, redexes occur only at positions of operation symbols. We have proved above that $\eta(t)$ is not a redex. Thus, there are no redexes in $\eta(t)$ above o and, by the induction hypothesis, the redex $\eta(t|_p)$ is outermost in $\eta(t)$ too. Claim number 3 vacuously holds.

$R' = ?$.

We have proved above that, in any narrowing derivation of $\eta(t)$ to a constructor-rooted term, a descendant of $\eta(t|_o)$ is narrowed to a constructor-rooted term. By the induction hypothesis, for any $\eta' \geq \sigma'$, $\eta'(\tau(t|_o))$ cannot be narrowed to a constructor-rooted term. Since $\eta \geq \sigma$, $\eta = \phi \circ \sigma$ for some substitution ϕ. Let $\eta' = \phi \circ \sigma' \geq \sigma'$ which implies $\eta = \eta' \circ \tau$ since $\sigma = \sigma' \circ \tau$. Thus, $\eta(t|_o) = \eta'(\tau(t|_o))$ cannot be narrowed to a constructor-rooted term. Hence, $\eta(t)$ cannot be narrowed to a constructor-rooted term. Claim number 2 vacuously holds. $\square$

We say that a narrowing derivation is *computed by* λ iff for each step $t \rightsquigarrow_{p, R, \sigma} t'$ of the derivation, (p, R, σ) belongs to $\lambda(t, \mathcal{T})$. The function λ implements our narrowing strategy as discussed next. The theorem shows (claim 2) that our strategy λ computes only outermost-needed narrowing steps. The theorem, however, does not show that the computation succeeds, i.e., a narrowing step is computed for any operation-rooted, hence expectedly narrowable, term. This requirement may seem essential, since to narrow a term "all the way" a strategy should compute a narrowing step, when one exists. Indeed, in incomplete rewrite systems, λ may fail to compute any narrowing step even when some step could be computed.

Example 9 Consider an incompletely defined operation, f, taking and returning a natural number.

$$f(0) \rightarrow 0$$

The term $t = f(s(f(0)))$ can be narrowed (actually rewritten, since it is ground) to its normal form, $f(s(0))$. The only redex position of t is $1 \cdot 1$, but λ on a minimal definitional tree of f returns the set $\{(1, ?, \{\})\}$.

The inability of λ to compute certain outermost-needed narrowing steps is a blessing in disguise. The theorem (claim 3) justifies giving up a narrowing attempt as soon as the failure to find a rule occurs—without further attempts to narrow t at other positions with the hope that a different rule might be found after other narrowing steps or that the position might be *deleted* [12] by another narrowing step. If $(p, ?, \sigma) \in \lambda(t, \mathcal{T})$, no equation having $\sigma(t)$ as one side can be solved. This is an opportunity for optimization. In fact $\sigma(t)$ may be narrowable at other positions different from p and an equation with $\sigma(t)$ as a side may even have an infinite search space. However, any amount of work applied toward finding a solution would be wasted.

Example 10 Consider the following term rewriting system for subtraction:

$$\begin{array}{rcll} X - 0 & \to & X & \mathrm{R}_1 \\ s(X) - s(Y) & \to & X - Y & \mathrm{R}_2 \end{array}$$

This term rewriting system is inductively sequential and a definitional tree, $\mathcal{T}$, of the operation "$-$" has an *exempt* node for the pattern $0 - s(X)$, i.e., the system is incomplete and $(\Lambda, ?, \{\}) \in \lambda(0-s(X), \mathcal{T})$. Therefore we can immediately stop the needed narrowing derivation of the equation $0 - s(X) \approx Y - Z$ although there exist infinitely many narrowing derivations for the right-hand side of this equation.

The definition of our outermost-needed narrowing strategy does not determine the computation space for a given inductively sequential rewrite system in a unique way. The concrete strategy depends on the definitional trees, and there is some freedom to construct these. For a discussion on how to compute definitional trees from rewrite rules and the implications of some non-deterministic choices of this computation see [2]. As we will show in Section 5, this does not affect the optimality of our strategy w.r.t. computed solutions. But in case of failing derivations, a definitional tree which is "unnecessarily large" could result in unnecessary derivation steps.

E.g., a minimal definitional tree of the operation "$-$" in Example 10 has an *exempt* node for the pattern $0 - s(X)$. However, Definition 12 also allows a definitional tree with a *branch* node for the pattern $0 - s(X)$ which has *exempt* nodes for the patterns $0 - s(0)$ and $0 - s(s(X_1))$. Our strategy would perform some unnecessary steps if this definitional tree were used for narrowing the term $0 - s(t)$, where t is an operation-rooted term. These unnecessary steps are avoided if all *branch* nodes in a definitional tree are useful, i.e., the tree is minimal.

However, the non-determinism of the trees of certain operations makes it possible that some work may be wasted when a narrowing derivation computed by λ terminates with a non-constructor term. The problem seems inevitable and is due to the inherent parallelism of certain operations, such as $\approx$ (this issue is discussed in some depth in [2, Display (8)]). The problem occurs only in terms with two or more outermost-needed narrowing positions, one of which cannot be narrowed to a constructor-rooted term.

4 Soundness and completeness

Outermost-needed narrowing is a sound and complete procedure to solve equations if we add the equality rules to narrow equations to *true*. The following proposition shows the equivalence between the reducibility to a same ground constructor term and the reducibility to *true* using the equality rules.

Proposition 1 *Let $\mathcal{R}$ be a term rewriting system without rules for $\approx$ and $\wedge$. Let $\mathcal{R}'$ be the system obtained by adding the equality rules to $\mathcal{R}$. The following propositions are equivalent for all terms t and t':*

1. *t and t' are reducible in $\mathcal{R}$ to a same ground constructor term.*
2. *$t \approx t'$ is reducible in $\mathcal{R}'$ to 'true'.*

Proof To show that claim 1 implies claim 2, consider a ground constructor term u such that $t \xrightarrow{*} u$ and $t' \xrightarrow{*} u$ using rules from $\mathcal{R}$. Hence, $t \approx t' \xrightarrow{*} u \approx u$ using rules from $\mathcal{R}'$. To show claim 2, it is sufficient to show $u \approx u \xrightarrow{*} true$ using the equality rules. This is done by induction on the structure of u. Base case: If u is a 0-ary constructor, say c, then $u \approx u$ can be directly reduced to *true* using the equality rule $c \approx c \rightarrow \mathit{true}$. Induction step: Let $u = c(t_1, \ldots, t_n)$. Then

$$u \approx u \;\rightarrow\; (t_1 \approx t_1) \wedge \cdots \wedge (t_n \approx t_n)$$

is a reduction step using the equality rule for the n-ary constructor c. By the induction hypothesis, $t_i \approx t_i \overset{*}{\rightarrow} \mathit{true}$ using the equality rules ($i = 1, \ldots, n$). Moreover, $\mathit{true} \wedge \cdots \wedge \mathit{true}$ can be reduced to *true* using the equality rule for $\wedge$.

To show that claim 2 implies claim 1, consider a reduction sequence $t \approx t' \overset{*}{\rightarrow} \mathit{true}$ using rules from $\mathcal{R}'$. We show the existence of a ground constructor term, u, such that $t \overset{*}{\rightarrow} u$ and $t' \overset{*}{\rightarrow} u$ using rules from $\mathcal{R}$ by induction on the number, say k, of $\approx$-rule applications in this reduction sequence. Base case ($k = 1$): There is exactly one application of a $\approx$-rule:

$$t \approx t' \;\overset{*}{\rightarrow}\; s \approx s' \;\rightarrow\; r \;\overset{*}{\rightarrow}\; \mathit{true}$$

r cannot have the symbol $\wedge$ at the root; otherwise, there must be further applications of a $\approx$-rule in the derivation $r \overset{*}{\rightarrow} \mathit{true}$. Hence, the applied $\approx$-rule is of the form $c \approx c \rightarrow \mathit{true}$ which implies claim 1. Induction step ($k > 1$): Then there is a first application of a $\approx$-rule:

$$t \approx t' \;\overset{*}{\rightarrow}\; s \approx s' \;\rightarrow\; r \;\overset{*}{\rightarrow}\; \mathit{true}$$

$r \neq \mathit{true}$; otherwise, there are no further applications of a $\approx$-rule in $r \overset{*}{\rightarrow} \mathit{true}$. Therefore, $s = c(t_1, \ldots, t_n)$, $s' = c(t'_1, \ldots, t'_n)$, and $r = (t_1 \approx t'_1) \wedge \cdots \wedge (t_n \approx t'_n)$. Since $r \overset{*}{\rightarrow} \mathit{true}$, an $\wedge$-rule must be applied to the root in this sequence, i.e., $r \overset{*}{\rightarrow} \mathit{true} \wedge r' \overset{*}{\rightarrow} \mathit{true}$. Thus, $t_1 \approx t'_1 \overset{*}{\rightarrow} \mathit{true}$ with at most $k-1$ $\approx$-rule applications. By the induction hypothesis, there is a ground constructor term u_1 such that $t_1 \overset{*}{\rightarrow} u_1$ and $t'_1 \overset{*}{\rightarrow} u_1$ using rules from $\mathcal{R}$. By a further induction on the arguments t_i, t'_i, we can show the existence of ground constructor terms $u_1, \ldots, u_n$ such that $t_i \overset{*}{\rightarrow} u_i$ and $t'_i \overset{*}{\rightarrow} u_i$ using rules from $\mathcal{R}$. Altogether, we obtain the derivations

$$\begin{array}{rcccc} t & \overset{*}{\rightarrow} & c(t_1, \ldots, t_n) & \overset{*}{\rightarrow} & c(u_1, \ldots, u_n) \\ t' & \overset{*}{\rightarrow} & c(t'_1, \ldots, t'_n) & \overset{*}{\rightarrow} & c(u_1, \ldots, u_n) \end{array}$$

using rules from $\mathcal{R}$. This implies claim 1. □

The soundness of outermost-needed narrowing is easy to prove since outermost-needed narrowing is a special case of general narrowing.

Theorem 2 (Soundness of outermost-needed narrowing) *Let $\mathcal{R}$ be an inductively sequential rewrite system extended by the equality rules. If $t \approx t' \overset{*}{\rightsquigarrow}_\sigma$ true is an outermost-needed narrowing derivation, then σ is a solution for $t \approx t'$.*

Proof If $t \approx t' \overset{*}{\rightsquigarrow}_\sigma \mathit{true}$, there exists a derivation

$$t \approx t' \rightsquigarrow_{p_1, R_1, \sigma_1} t_1 \rightsquigarrow_{p_2, R_2, \sigma_2} \cdots \rightsquigarrow_{p_n, R_n, \sigma_n} t_n$$

such that $t_n = \mathit{true}$ and $\sigma = \sigma_n \circ \cdots \circ \sigma_1$. By induction on the number n of narrowing steps, it is easy to prove that $\sigma(t \approx t') \overset{*}{\rightarrow} \mathit{true}$. By Proposition 1, this implies that $\sigma(t)$ and $\sigma(t')$ are reducible to a same ground constructor term without using the equality rules. By Definition 7, σ is a solution for $t \approx t'$. □

In order to prove completeness of the outermost-needed narrowing strategy, we lift the completeness result for the corresponding rewrite strategy [2] to narrowing derivations. For this purpose, we recall the definition of the outermost-needed rewrite strategy for inductively sequential systems. Similarly to λ, this rewrite strategy is implemented by a function, φ, that takes two arguments: an operation-rooted term, t, and a definitional tree, $\mathcal{T}$, of the root of t (the definition of φ is a slightly modified version of the definition given in [2] extended to non-ground terms). Throughout an interleaved descent down both t and $\mathcal{T}$, φ computes a position p and, whenever possible, a rule R such that the rewriting of t at p by means of R is outermost-needed.

Definition 14 The function φ takes two arguments, an operation-rooted term t and a *pdt* $\mathcal{T}$ such that $\mathit{pattern}(\mathcal{T}) \leq t$. The function φ yields a pair, (p, R), where p is a position of t and R is either a rewrite rule or the distinguished symbol "?." Thus, let t be a term and $\mathcal{T}$ be a *pdt* in the domain of φ. The function φ is defined by structural induction on t with a nested structural induction on $\mathcal{T}$ as follows.

$$\varphi(t,\mathcal{T}) = \begin{cases} (\Lambda, R) & \text{if } \mathcal{T} = \mathit{rule}(\pi, R); \\ (\Lambda, ?) & \text{if } \mathcal{T} = \mathit{exempt}(\pi); \\ \varphi(t, \mathcal{T}_i) & \text{if } \mathcal{T} = \mathit{branch}(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k) \text{ and } \mathit{pattern}(\mathcal{T}_i) \leq t, \text{ for some } i; \\ (o \cdot p, R) & \text{if } \mathcal{T} = \mathit{branch}(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad t|_o \text{ is operation-rooted,} \\ & \quad \mathcal{T}' \text{ is a definitional tree of the root of } t|_o, \text{ and} \\ & \quad \varphi(t|_o, \mathcal{T}') = (p, R). \\ (o, ?) & \text{if } \mathcal{T} = \mathit{branch}(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k) \text{ and } t|_o \text{ is a variable} \end{cases}$$

The function φ is well-defined in the third case since, by the definition of *pdt*, a proper sub*pdt* $\mathcal{T}_i$ of $\mathcal{T}$ with $\mathit{pattern}(\mathcal{T}_i) \leq t$ must uniquely exist iff $t|_o$ is constructor-rooted. Similarly, φ is well-defined in the fourth case since $t|_o$ is a proper subterm of t and $\mathit{pattern}(\mathcal{T}') \leq t|_o$ by the definition of *pdt*.

The following theorem, which compacts various results of [2], shows that the function φ computes outermost-needed redexes. The proof parallels that of Theorem 1.

Theorem 3 *Let $\mathcal{R}$ be an inductively sequential rewrite system, t an operation-rooted term, $\mathcal{T}$ a definitional tree of the root of t, and $\varphi(t, \mathcal{T}) = (p, R)$.*

1. *In any reduction sequence of t to a constructor-rooted term, a descendant of $t|_p$ is reduced to a constructor-rooted term.*
2. *If R is a rule of $\mathcal{R}$, then $t|_p$ is an outermost-needed redex of t matched by R.*
3. *If $R = ?$, then t cannot be reduced to a constructor-rooted term.*

Proof We prove by Noetherian induction on $\prec$ the claim generalized by considering $\mathcal{T}$ a subtree of a definitional tree of the root of t such that $\mathit{pattern}(\mathcal{T}) \leq t$. We consider the cases of the definition of φ.

Base case: consider $(t, \mathcal{T})$ where t is an operation-rooted term and $\mathcal{T} = \mathit{leaf}(\pi)$ for some pattern π. We consider the two subcases of the definition of φ for *leaf* nodes:

$\mathcal{T} = \mathit{rule}(\pi, R')$, for some pattern π and rule R'.

In this case, $(p, R) = (\Lambda, R')$. Since t is operation-rooted and is a descendant of itself, claim number 1 trivially holds. By the hypothesis, $\pi \leq t$. By the definition of definitional tree, R is a rule whose left-hand side is equal to π. Thus, t is a redex, it is matched by R, and it is obviously outermost. Since $\mathcal{R}$ is orthogonal (by Lemma 3), its redex schemes do not overlap; consequently, R keeps matching any descendant of t obtained by reductions strictly below Λ. Thus, t is a needed redex. Claim number 3 vacuously holds.

$\mathcal{T} = \mathit{exempt}(\pi)$, for some pattern π.

In this case, $(p, R) = (\Lambda, ?)$. By the hypothesis, $\pi \leq t$. We could extend $\mathcal{R}$ by changing the *exempt* node into a *rule* node in which the left-hand side of the rule is obviously π and the right-hand side is arbitrary. Thus, similar to the previous case, π keeps matching any descendant of t obtained by reductions strictly below the root. Thus, by Lemma 2, there exists no rule in $\mathcal{R}$ for a reduction of t at the root. Thus, t cannot be reduced to a constructor-rooted term which implies claim number 3 and, trivially, claim number 1. Claim number 2 vacuously holds.

Induction step: consider $(t, \mathcal{T})$ where t is an operation-rooted term and $\mathcal{T} = \mathit{branch}(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, for some pattern π, position o, and *pdt*s $\mathcal{T}_1, \ldots, \mathcal{T}_k$, for some $k > 0$. We consider three exhaustive (and mutually exclusive) cases for $t|_o$:

$t|_o$ is constructor-rooted.

By the definition of *pdt*, there exists some i in $\{1, \ldots, k\}$ such that $\mathit{pattern}(\mathcal{T}_i) \leq t$. By the definition of φ, $\varphi(t, \mathcal{T}) = \varphi(t, \mathcal{T}_i)$. By the induction hypothesis, all the claims hold already for $\varphi(t, \mathcal{T}_i)$ and they are independent of $\mathcal{T}_i$.

$t|_o$ is operation-rooted.

Let $\mathcal{T}'$ be a definitional tree of the root of $t|_o$. Let $\varphi(t|_o, \mathcal{T}') = (p', R')$, where p' is a position of $t|_o$ and R' is either a rule or "?." In this case, $(p, R) = (o \cdot p', R')$ and $(t|_o)|_{p'} = t|_{o \cdot p'} = t|_p$. By the hypothesis, $\pi \leq t$. By Lemma 2, any rule that might reduce t at the root is represented by a leaf of some $\mathcal{T}_i$ with $1 \leq i \leq k$ since the left-hand side of any other rule does not unify with π. If $l \to r$ is a rule represented by a leaf of $\mathcal{T}_i$ then, by Lemma 1, $\mathit{pattern}(\mathcal{T}_i) \leq l$. Thus, by the definition of definitional tree, l has a constructor symbol at position o. However, the case being considered assumes that t does not have a constructor symbol at position o. Hence, in any reduction sequence of t that includes a reduction at the root a descendant of $t|_o$ must be reduced to a constructor-rooted term. Since t is operation-rooted, in any reduction sequence of t to a constructor-rooted term, a descendant of t is reduced at the root. Consequently, a descendant of $t|_o$ is reduced to a constructor-rooted term. By the induction hypothesis, in any reduction sequence of $t|_o$ to a constructor-rooted term, a descendant of $t|_p$ is reduced to a constructor-rooted term. Thus, claim number 1 holds by transitivity.

We consider the two cases for R':

R' is a rule.

By the induction hypothesis, $t|_p$ is an outermost-needed redex of $t|_o$ matched by R'; hence, $t|_p$ is a redex of t matched by R'. The need of $t|_p$ with respect to t is an immediate consequence of claim number 1. By the hypothesis, $\pi \leq t$, and by the definition of definitional tree, π is a pattern and o is a position of π. These conditions imply that there is only one operation symbol in t above o: the root of t. In constructor-based systems, redexes occur only at positions of operation symbols.

We have just proved that t is not a redex. Thus, there are no redexes in t above o and, by the induction hypothesis, the redex $t|_p$ is outermost in t too.

$R' = ?$.

By claim number 1, in any reduction sequence of t to a constructor-rooted term, a descendant of $t|_p$ is reduced to a constructor-rooted term. By the induction hypothesis, $t|_p$ cannot be reduced to a constructor-rooted term. Thus, t cannot be reduced to a constructor-rooted term.

$t|_o$ is a variable.

In this case, $(p, R) = (o, ?)$. The proofs of claims number 1 and 3 are similar to those of the previous case, but do not require induction hypotheses. Claim number 2 vacuously holds. □

The following proposition shows that the strategy λ behaves as φ on ground terms.

Proposition 2 *Let $\mathcal{R}$ be an inductively sequential rewrite system. Let t be a ground operation-rooted term, $\mathcal{T}$ be a definitional tree of the root of t. Then, there exists a substitution σ such that $\lambda(t, \mathcal{T}) = \{(p, R, \sigma)\}$ and $\varphi(t, \mathcal{T}) = (p, R)$, where p is some position, R is some rewrite rule or the distinguished symbol "?," and σ is some substitution.*

Proof We prove by Noetherian induction on $\prec$ the claim generalized by considering $\mathcal{T}$ a subtree of a definitional tree of the root of t satisfying the condition $\mathit{pattern}(\mathcal{T}) \leq t$. We consider the cases of the definitions of λ and φ:

Base case: consider $(t, \mathcal{T})$ where t is an operation-rooted term and $\mathcal{T} = \mathit{leaf}(\pi)$, for some pattern π. We consider the two subcases of the definitions of λ and φ for *leaf* nodes.

$\mathcal{T} = \mathit{rule}(\pi, R')$, for some pattern π and rule R'.

In this case, $\varphi(t, \mathcal{T}) = (\Lambda, R') = (p, R)$ and $\lambda(t, \mathcal{T}) = \{(\Lambda, R', \mathit{mgu}(t, \pi))\} = \{(p, R, \sigma)\}$.

$\mathcal{T} = \mathit{exempt}(\pi)$, for some pattern π.

In this case $\varphi(t, \mathcal{T}) = (\Lambda, ?) = (p, R)$ and $\lambda(t, \mathcal{T}) = \{(\Lambda, ?, \mathit{mgu}(t, \pi))\} = \{(p, R, \sigma)\}$. Thus the claim holds.

Induction step: consider $(t, \mathcal{T})$ where t is an operation-rooted term and $\mathcal{T} = \mathit{branch}(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, for some pattern π, position o, and *pdt*s $\mathcal{T}_1, \ldots, \mathcal{T}_k$, for some $k > 0$. We consider three exhaustive (and mutually exclusive) cases for $t|_o$:

$t|_o$ is constructor-rooted.

By definition of *pdt*, there exists some i in $\{1, \ldots, k\}$ such that $\mathit{pattern}(\mathcal{T}_i)$ and t unify. Since t is ground, we have $\mathit{pattern}(\mathcal{T}_i) \leq t$ and for all j in $\{1, \ldots, k\}$ such that $j \neq i$, $\mathit{pattern}(\mathcal{T}_j)$ and t do not unify. Therefore, by the definition of φ, $\varphi(t, \mathcal{T}) = \varphi(t, \mathcal{T}_i)$ and by the definition of λ, $\lambda(t, \mathcal{T}) = \lambda(t, \mathcal{T}_i)$. Thus, by the induction hypothesis the claim holds.

$t|_o$ is operation-rooted.

By the hypothesis, $\pi \leq t$. Hence, t and π unify. Let $\tau = \mathit{mgu}(t, \pi)$. Since t is ground, $\tau(t) = t$. By the definition of *pdt*, t and $\mathit{pattern}(\mathcal{T}_i)$ do not unify for each i in $\{1, \ldots, k\}$ since $t|_o$ is operation-rooted, whereas $\mathit{pattern}(\mathcal{T}_i)$ has a constructor symbol at position o. Let $\mathcal{T}'$ be a definitional tree of the root of $t|_o$. By the definition of φ, $\varphi(t, \mathcal{T}) = (o \cdot p', R')$ where $(p', R') = \varphi(t|_o, \mathcal{T}')$. Likewise, by the definition of λ, $\lambda(t, \mathcal{T}) = \{(o \cdot$

$p'', R'', \sigma'' \circ \tau) \mid (p'', R'', \sigma'') \in \lambda(\tau(t|_o), \mathcal{T}')\}$. Since $t|_o$ contains fewer operation symbols than t, we deduce by induction hypotheses that there exists a substitution σ'' such that $\lambda(\tau(t|_o), \mathcal{T}') = \lambda(t|_o, \mathcal{T}') = \{(p', R', \sigma'')\}$. Thus, $\lambda(t, \mathcal{T}) = \{(o \cdot p', R', \sigma'' \circ \tau)\}$ and the claim holds.

$t|_o$ is a variable: This case cannot occur since t is ground. □

The following lemma shows the close ties between φ and λ, which are instrumental to lift outermost-needed reduction sequences to corresponding narrowing derivations. This will allow us to prove the completeness of the outermost-needed narrowing strategy.

Lemma 4 *Let $\mathcal{R}$ be an inductively sequential rewrite system. Let t be an operation-rooted term, $\mathcal{T}$ be a definitional tree of the root of t, and σ be a constructor substitution. If $\sigma(t) \to_{p,R} t'$ with $(p, R) = \varphi(\sigma(t), \mathcal{T})$, then there exists a substitution θ such that*

1. $(p, R, \theta) \in \lambda(t, \mathcal{T})$
2. $\theta \le \sigma[\mathcal{V}ar(t)]$

Proof We prove by Noetherian induction on $\prec$ the claim generalized by considering $\mathcal{T}$ a subtree of a definitional tree of the root of t. We consider the cases of the definition of φ:

Base case: consider $(t, \mathcal{T})$ where t is an operation-rooted term and $\mathcal{T} = \mathit{leaf}(\pi)$, for some pattern π. We consider the two subcases of the definition of φ for *leaf* nodes.

$\mathcal{T} = \mathit{rule}(\pi, R')$, for some pattern π and rule R'.

In this case, $(p, R) = (\Lambda, R')$ and $\pi \le \sigma(t)$. This implies the existence of a substitution ϕ with $\phi(\pi) = \sigma(t)$. Hence, π and t are unifiable (we assume that π and t are variable disjoint. Otherwise, take a new variant of the definitional tree) and there exists a most general unifier θ of π and t with $\theta \le \sigma[\mathcal{V}ar(t)]$. By the definition of λ, $(p, R, \theta) \in \lambda(t, \mathcal{T})$.

$\mathcal{T} = \mathit{exempt}(\pi)$: This case cannot occur since $R \neq ?$.

Induction step: consider $(t, \mathcal{T})$ where t is an operation-rooted term and $\mathcal{T} = \mathit{branch}(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, for some pattern π, position o, and *pdt*s $\mathcal{T}_1, \ldots, \mathcal{T}_k$, for some $k > 0$. We consider three exhaustive (and mutually exclusive) cases for $t|_o$:

$\sigma(t)|_o$ is constructor-rooted.

By the definition of *pdt*, there exists some i in $\{1, \ldots, k\}$ such that $\mathit{pattern}(\mathcal{T}_i) \le \sigma(t)$. By the definition of φ, $\varphi(\sigma(t), \mathcal{T}) = \varphi(\sigma(t), \mathcal{T}_i)$. By the induction hypothesis, $(p, R, \theta) \in \lambda(t, \mathcal{T}_i)$ and $\theta \le \sigma[\mathcal{V}ar(t)]$. By the definition of λ (note that $\mathit{pattern}(\mathcal{T}_i)$ and t unify), $(p, R, \theta) \in \lambda(t, \mathcal{T})$.

$\sigma(t)|_o$ is operation-rooted.

By the definition of φ, $\pi \le \sigma(t)$. $\sigma(t)$ and $\mathit{pattern}(\mathcal{T}_i)$ do not unify for each i in $\{1, \ldots, k\}$ since $\sigma(t)|_o$ is operation-rooted, but $\mathit{pattern}(\mathcal{T}_i)$ has a constructor symbol at position o. Let $\mathcal{T}'$ be a definitional tree of the root of $\sigma(t)|_o$ and $\varphi(\sigma(t)|_o, \mathcal{T}') = (p', R')$. By the definition of φ, $(p, R) = (o \cdot p', R')$. Since $\pi \le \sigma(t)$, there exists a most general unifier τ of π and $\sigma(t)$ with $\tau \le \sigma[\mathcal{V}ar(t)]$ (we assume that π and t are variable disjoint, otherwise take a new variant of the definitional tree). $\tau_{|\mathcal{V}ar(t)}$ is a constructor substitution since π is a linear pattern and t is operation-rooted. Let σ' be a constructor substitution such that $\sigma' \circ \tau = \sigma[\mathcal{V}ar(t)]$. Since σ is a constructor substitution, o is a position of t, and $t|_o$ is operation-rooted. Moreover, $\sigma(t) \rightarrow_{p,R} t'$ implies $\sigma'(\tau(t|_o)) \rightarrow_{p',R'} t'|_o$. Since o is different from the root position, t is operation-rooted, and $\tau_{|\mathcal{V}ar(t)}$ is a constructor substitution, $\tau(t|_o)$ has fewer occurrences of defined operation symbols than t. Hence, by induction hypothesis applied to $(\tau(t|_o), \mathcal{T}')$ and σ', there exists a substitution θ' such that $(p', R', \theta') \in \lambda(\tau(t|_o), \mathcal{T}')$ and $\theta' \leq \sigma'[\mathcal{V}ar(\tau(t|_o))]$. By the definition of λ, $(o \cdot p', R', \theta' \circ \tau) \in \lambda(t, \mathcal{T})$, i.e., $(p, R, \theta' \circ \tau) \in \lambda(t, \mathcal{T})$. $\theta' \leq \sigma'[\mathcal{V}ar(\tau(t|_o))]$ implies $\theta' \leq \sigma'[\mathcal{V}ar(\tau(t))]$ since θ' instantiates only variables from $\tau(t|_o)$ and new variables of the definitional tree. Hence, $\theta' \circ \tau \leq \sigma' \circ \tau[\mathcal{V}ar(t)]$ which is equivalent to $\theta' \circ \tau \leq \sigma[\mathcal{V}ar(t)]$.

$\sigma(t)|_o$ is a variable: This case cannot occur since $R \neq ?$. □

The following lemma shows how to lift an outermost-needed reduction step to an outermost-needed narrowing step.

Lemma 5 *Let $\mathcal{R}$ be an inductively sequential rewrite system. Let σ be a constructor substitution, V be a finite set of variables, t be an operation-rooted term with $\mathcal{V}ar(t) \subseteq V$, and $\mathcal{T}$ be a definitional tree of the root of t. If $\sigma(t) \rightarrow_{p,R} s$ with $(p, R) = \varphi(\sigma(t), \mathcal{T})$, then there exist an outermost-needed narrowing step $t \rightsquigarrow_{p,R,\theta} t'$ and a substitution σ' such that $(p, R, \theta) \in \lambda(t, \mathcal{T})$, $\sigma'(t') = s$ and $\sigma' \circ \theta = \sigma[V]$.*

Proof Let R be $l \rightarrow r$. By Lemma 4, there is a triple $(p, R, \theta) \in \lambda(t, \mathcal{T})$ with $\theta \leq \sigma[\mathcal{V}ar(t)]$. Then there exists σ' such that $\sigma' \circ \theta = \sigma[V]$ (since θ instantiates only variables occurring in t and in the definitional trees, we assume that $\theta(x) = x$ for all $x \in V - \mathcal{V}ar(t)$ by taking appropriate variants of the definitional trees). By claim 2 of Theorem 1, $t \rightsquigarrow_{p,R,\theta} t'$ is an outermost-needed narrowing step and $\theta(t)|_p = \theta(l)$. Hence $\sigma(t)|_p = \sigma'(\theta(t))|_p = \sigma'(\theta(l))$ which implies $\sigma(t)[\sigma'(\theta(r))]_p = s$. Finally, $\sigma'(t') = \sigma'(\theta(t[r]_p)) = \sigma'(\theta(t))[\sigma'(\theta(r))]_p = \sigma(t)[\sigma'(\theta(r))]_p = s$. □

Outermost-needed narrowing instantiates variables to constructor terms. Thus, we only show that outermost-needed narrowing is complete for constructor substitutions as solutions of equations. This is not a limitation in practice, since more general solutions would contain unevaluated or undefined expressions. This is not a limitation with respect to related work, since most general narrowing is known to be complete only for normalizable solutions [51] (which can be seen as semantically equivalent to irreducible solutions), and lazy narrowing is complete only for constructor substitutions [23, 55]. Incidentally, we also believe that needed narrowing is complete for the entire class of the orthogonal rewrite systems w.r.t. irreducible substitutions, although neither needed rewriting nor needed narrowing steps are computable for this class of rewrite systems. The following theorem shows the completeness of our strategy, λ, and consequently of outermost-needed narrowing, for inductively sequential rewrite systems:

Theorem 4 (Completeness of outermost-needed narrowing) *Let $\mathcal{R}$ be an inductively sequential rewrite system extended by the equality rules. Let σ be a constructor substitution that is a solution of an equation $t \approx t'$ and V be a finite set of variables containing $\mathcal{V}ar(t) \cup \mathcal{V}ar(t')$. Then there exists a derivation $t \approx t' \overset{*}{\rightsquigarrow}_{\sigma'} true$ computed by λ such that $\sigma' \leq \sigma[V]$.*

Proof By Definition 7, there exists a ground constructor term, say u, such that $\sigma(t \approx t') \overset{*}{\rightarrow} u \approx u$. Since $\mathcal{R}$ is extended by the equality rules, $\sigma(t \approx t') \overset{*}{\rightarrow} true$ by Proposition 1. Consider the following reduction sequence

$$s_0 \rightarrow_{p_1,R_1} s_1 \rightarrow_{p_2,R_2} s_2 \rightarrow_{p_3,R_3} \cdots$$

where $s_0 = \sigma(t \approx t')$, $(p_{i+1}, R_{i+1}) = \varphi(s_i, \mathcal{T}_i)$ and $\mathcal{T}_i$ is a definitional tree of the root of s_i for $i = 0, 1, 2, \ldots$. The following claims are easy to show by induction on the derivation steps in this sequence:

1. s_i has a constructor-rooted normal form (*true*).
2. If $s_i \neq \mathit{true}$, then the root of s_i is the operation symbol $\wedge$ or $\approx$ (by the definition of equality rules).
3. If $s_i \neq \mathit{true}$, then $R_{i+1} \neq ?$ (claim 3 of Theorem 3) and $s_i|_{p_{i+1}}$ is an outermost-needed redex (claim 2 of Theorem 3).

Hence, the reduction sequence is well-defined and outermost-needed (as long as $s_i \neq \mathit{true}$). Since repeated rewriting of needed redexes in a term computes the term's normal form, if it exists [41], the sequence is finite and $s_n = \mathit{true}$ is the final term for some $n > 0$. We will show by induction on n that there exists a corresponding outermost-needed narrowing derivation

$$t \approx t' \leadsto_{p_1, R_1, \sigma_1} t_1 \leadsto_{p_2, R_2, \sigma_2} \cdots \leadsto_{p_n, R_n, \sigma_n} t_n$$

such that $t_n = \mathit{true}$ and $\sigma_n \circ \cdots \circ \sigma_1 \leq \sigma[V]$.

$n = 1$: If we apply Lemma 5 to the reduction step $s_0 \rightarrow_{p_1, R_1} s_1$, we obtain an outermost-needed narrowing step $t \approx t' \leadsto_{p_1, R_1, \sigma_1} t_1$ and a substitution σ' such that $\sigma' \circ \sigma_1 = \sigma[V]$ and $\sigma'(t_1) = s_1 = \mathit{true}$. Hence, $\sigma_1 \leq \sigma[V]$ and $t_1 = \mathit{true}$ by the definition of equality rules.

$n > 1$: By Lemma 5 applied to the first reduction step, there exist an outermost-needed narrowing step $t \approx t' \leadsto_{p_1, R_1, \sigma_1} t_1$ and a substitution σ' such that $\sigma' \circ \sigma_1 = \sigma[V]$ and $\sigma'(t_1) = s_1$. Let $V_1 = \{y \in \mathcal{V}ar(\sigma_1(x)) \mid x \in V\}$ (note that $\mathcal{V}ar(t_1) \subseteq V_1$). Applying the induction hypothesis to V_1, σ' (note that $\sigma'_{|V_1}$ is a constructor substitution since σ is a constructor substitution) and the derivation

$$s_1 \rightarrow_{p_2, R_2} \cdots \rightarrow_{p_n, R_n} s_n$$

yields an outermost-needed narrowing derivation

$$t_1 \leadsto_{p_2, R_2, \sigma_2} \cdots \leadsto_{p_n, R_n, \sigma_n} t_n$$

with $t_n = \mathit{true}$ and $\sigma_n \circ \cdots \circ \sigma_2 \leq \sigma'[V_1]$. Combining that with the first narrowing step, we obtain the required outermost-needed narrowing derivation with $\sigma_n \circ \cdots \circ \sigma_1 \leq \sigma[V]$ since $\sigma' \circ \sigma_1 = \sigma[V]$. □

The theorem justifies our earlier remark on the relationship between completeness and anticipated substitutions. Any anticipated substitution of a needed narrowing step is irrelevant or would eventually be done later in the derivation; thus, it does not affect the completeness. Anticipating substitutions is appealing even without the benefits related to the need of a step, since less general substitutions are likely to yield a smaller search space to compute the same set of solutions.

5 Optimality

In Section 3, we showed that our strategy computes only necessary steps. We now strengthen this characterization by showing that our strategy computes only necessary derivations of minimum cost. First of all, we show that no redundant derivation is computed by λ. For this purpose, we need some technical definitions and results that we give below.

Definition 15 Let $\mathcal{R}$ be a term rewriting system. Let t and s be two terms. We write $t =_{\mathcal{R}} s$ iff $t \overset{*}{\leftrightarrow} s$. Let σ_1 and σ_2 be two substitutions and V a set of variables. We write $\sigma_1 =_{\mathcal{R}} \sigma_2[V]$ iff $\sigma_1(x) =_{\mathcal{R}} \sigma_2(x)$ for all $x \in V$, and likewise we write $\sigma_1 \neq_{\mathcal{R}} \sigma_2[V]$ iff $\sigma_1(x) \neq_{\mathcal{R}} \sigma_2(x)$ for some $x \in V$. We write $\sigma_1 \leq_{\mathcal{R}} \sigma_2[V]$ iff there exists a substitution θ such that $\theta \circ \sigma_1 =_{\mathcal{R}} \sigma_2[V]$. We say that σ_1 and σ_2 are *incomparable on* V iff neither $\sigma_1 \leq_{\mathcal{R}} \sigma_2[V]$ nor $\sigma_2 \leq_{\mathcal{R}} \sigma_1[V]$. σ_1 and σ_2 are called *disjoint on* V iff $\theta_1 \circ \sigma_1 \neq_{\mathcal{R}} \theta_2 \circ \sigma_2[V]$ for all substitutions θ_1 and θ_2.

The incomparability of substitutions has been used in unification theory [64] in order to characterize minimal sets of solutions. When they exist, such minimal sets are unique (up to $=_{\mathcal{R}}$). Nevertheless, incomparable substitutions might have a common instance and, therefore, they do not describe disjoint regions of the solution space. For instance, the substitutions $\{x \mapsto 0\}$ and $\{y \mapsto 0\}$ are incomparable on $\{x, y\}$ but the substitution $\{x \mapsto 0, y \mapsto 0\}$ is described by both substitutions. However, this is not the case for disjoint substitutions since, by definition, they describe independent regions of the solution space. Thus, disjointness is a stronger notion than incomparability and we will show that the solutions computed by our strategy, λ, are disjoint. First we show that the property of disjointness is indeed stronger than incomparability.

Proposition 3 *Let σ_1 and σ_2 be two substitutions and V a set of variables. If σ_1 and σ_2 are disjoint on V, then σ_1 and σ_2 are incomparable on V.*

Proof Assume that σ_1 and σ_2 are not incomparable on V. That is to say, either $\sigma_1 \leq_{\mathcal{R}} \sigma_2[V]$ or $\sigma_2 \leq_{\mathcal{R}} \sigma_1[V]$. Suppose without loss of generality that $\sigma_1 \leq_{\mathcal{R}} \sigma_2[V]$. By definition of the preordering $\leq_{\mathcal{R}}$, there exists a substitution θ_1 such that $\theta_1 \circ \sigma_1 =_{\mathcal{R}} \sigma_2[V]$. Thus, by Definition 15, σ_1 and σ_2 are not disjoint on V. □

The two following technical propositions are used in subsequent proofs.

Proposition 4 *Let $\mathcal{R}$ be an inductively sequential rewrite system. Let c_1 and c_2 be two constructor symbols. If $c_1(t_1, \ldots, t_n) =_{\mathcal{R}} c_2(u_1, \ldots, u_m)$, where $n, m \geq 0$ and t_i and u_j are terms for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, then $c_1 = c_2$ (and thus, $n = m$) and $t_i =_{\mathcal{R}} u_i$, for $i = 1, \ldots, n$.*

Proof Since inductively sequential rewrite systems are orthogonal (cf. Lemma 3), they are confluent and Church-Russer (a TRS $\mathcal{R}$ is *Church-Russer* iff for all terms t_1 and t_2, $t_1 =_{\mathcal{R}} t_2$ implies the existence of a term t_3 with $t_1 \overset{*}{\rightarrow} t_3$ and $t_2 \overset{*}{\rightarrow} t_3$). From the Church-Russer property of $\mathcal{R}$ and the statement $c_1(t_1, \ldots, t_n) =_{\mathcal{R}} c_2(u_1, \ldots, u_m)$, we infer the existence of a term T such that $c_1(t_1, \ldots, t_n) \overset{*}{\rightarrow} T$ and $c_2(u_1, \ldots, u_m) \overset{*}{\rightarrow} T$. Since inductively sequential rewrite systems are constructor-based, no descendants of the terms $c_1(t_1, \ldots, t_n)$ and $c_2(u_1, \ldots, u_m)$ can be rewritten at the root. Therefore, $c_1 = c_2$ (and thus $n = m$) and the term T is of the form $T = c_1(w_1, \ldots, w_n)$ such that for all $i = 1, \ldots, n$, $t_i \overset{*}{\rightarrow} w_i$ and $u_i \overset{*}{\rightarrow} w_i$. The last statement shows that for all $i = 1, \ldots, n$, $t_i =_{\mathcal{R}} u_i$. □

Proposition 5 *Let $\mathcal{R}$ be an inductively sequential rewrite system. Let t be a constructor term and σ_1 and σ_2 arbitrary substitutions. Then*

$$\sigma_1(t) =_{\mathcal{R}} \sigma_2(t) \tag{1}$$

implies for all variables $y \in \mathcal{V}ar(t)$,

$$\sigma_1(y) =_{\mathcal{R}} \sigma_2(y) \tag{2}$$

Proof The proof is by structural induction on t. Base case: t is either a constant or a variable. In the first subcase, (2) is vacuously true, in the second subcase, (2) is a direct consequence of (1). Induction step: Let $t = c(t_1, \dots, t_n)$, for some constructor symbol c and constructor terms $t_1, \dots, t_n$. By definition of substitution, $c(\sigma_1(t_1), \dots, \sigma_1(t_n)) = \sigma_1(t)$ and $c(\sigma_2(t_1), \dots, \sigma_2(t_n)) = \sigma_2(t)$. From Proposition 4 and $c(\sigma_1(t_1), \dots, \sigma_1(t_n)) =_{\mathcal{R}} c(\sigma_2(t_1), \dots, \sigma_2(t_n))$, we deduce that $\sigma_1(t_i) =_{\mathcal{R}} \sigma_2(t_i)$ for $i = 1, \dots, n$. Thus, the claim is immediate from the induction hypothesis. $\Box$

Proposition 6 *Let $\mathcal{R}$ be an inductively sequential rewrite system extended by the equality rules, e an equation to solve and $V = \mathcal{V}ar(e)$. Let $e \overset{+}{\rightsquigarrow}_\sigma e'$ be a derivation computed by λ. Then, $\sigma_{|V}$ is a constructor substitution.*

Proof The proof is by induction on the length n of $e \overset{+}{\rightsquigarrow}_\sigma e'$. Base case: $n = 1$. In this case $e \rightsquigarrow_{p,l \to r,\sigma} e'$ with $(p, l \to r, \sigma) \in \lambda(e, \mathcal{T})$, where $\mathcal{T}$ is a definitional tree of the root of e. Since the patterns of definitional trees are linear patterns with fresh variables, $\sigma_{|V}$ is a constructor substitution. Induction step: Consider now the derivation $e \rightsquigarrow_{p,l \to r,\sigma_1} e_1 \overset{+}{\rightsquigarrow}_\sigma e'$. By the induction hypothesis, $\sigma_{|\mathcal{V}ar}(e_1)$ and $\sigma_{1|V}$ are constructor substitutions. Thus $(\sigma \circ \sigma_1)_{|V}$ is a constructor substitution. $\Box$

Proposition 7 *Let $t_0 \rightsquigarrow_{p_1,l_1 \to r_1,\sigma_1} t_1 \dots \rightsquigarrow_{p_n,l_n \to r_n,\sigma_n} t_n$ be a narrowing derivation. Then, $\forall x \in \mathcal{V}ar(t_n)$, $\exists\, y \in \mathcal{V}ar(t_0)$ such that $x \in \mathcal{V}ar(\sigma_n \circ \dots \circ \sigma_1(y))$.*

Proof Note that $\sigma_n \circ \dots \circ \sigma_1(t_0) \overset{*}{\rightarrow} t_n$ (compare proof of Theorem 2). Since reduction steps do not introduce new variables, $x \in \mathcal{V}ar(\sigma_n \circ \dots \circ \sigma_1(t_0))$ for all $x \in \mathcal{V}ar(t_n)$ which implies the claim. $\Box$

Proposition 8 *Let $\mathcal{R}$ be an inductively sequential rewrite system. Let t be an operation-rooted term, $V = \mathcal{V}ar(t)$ and (p_1, R_1, σ_1) and (p_2, R_2, σ_2) two distinct triples in $\lambda(t, \mathcal{T})$. Then, σ_1 and σ_2 are disjoint on V.*

Proof The proof is by Noetherian induction on $\prec$. We consider the cases of the definition of λ.

Base case: consider $(t, \mathcal{T})$ where t is an operation-rooted term and $\mathcal{T} = \textit{leaf}(\pi)$, for some pattern π. There are no distinct triples in $\lambda(t, \mathcal{T})$ and the claim vacuously holds.

Induction step: consider $(t, \mathcal{T})$ where t is an operation-rooted term and $\mathcal{T} = \textit{branch}(\pi, o, \mathcal{T}_1, \dots, \mathcal{T}_k)$, for some pattern π, position o, and *pdt*s $\mathcal{T}_1, \dots, \mathcal{T}_k$, for some $k > 0$. We consider three exhaustive (and mutually exclusive) cases for $t|_o$:

$t|_o$ is a variable, say x.

In this case t and $\textit{pattern}(\mathcal{T}_i)$ unify for all $i = 1, \dots, k$. By the induction hypothesis, for every i, the substitutions of distinct triples in $\lambda(t, \mathcal{T}_i)$ are disjoint on V. Moreover, if $(p_i, R_i, \sigma_i) \in \lambda(t, \mathcal{T}_i)$ and $(p_j, R_j, \sigma_j) \in \lambda(t, \mathcal{T}_j)$ with $i \neq j$, then by definition of λ the roots of $\sigma_i(x)$ and $\sigma_j(x)$ are different constructors. So, for all substitutions θ_1 and θ_2, the roots of $\theta_1(\sigma_i(x))$ and $\theta_2(\sigma_j(x))$ are different constructors too. Hence by Proposition 4 we have $\theta_1(\sigma_i(x)) \neq_{\mathcal{R}} \theta_2(\sigma_j(x))$ which shows that the substitutions σ_i and σ_j are disjoint on V. Thus, the claim holds.

$t|_o$ is a constructor-rooted.

By the definition of *pdt*, there exists one i in $\{1, \ldots, k\}$ such that *pattern*$(\mathcal{T}_i)$ and t unify. By the definition of λ, $\lambda(t, \mathcal{T}) = \lambda(t, \mathcal{T}_i)$. By the induction hypothesis, the claim holds for $\lambda(t, \mathcal{T}_i)$ and thus for $\lambda(t, \mathcal{T})$ too.

$t|_o$ is operation-rooted.

By the definition of λ, π and t unify. Let $\tau = mgu(t, \pi)$. Since $t|_o$ is operation-rooted, so is $\tau(t|_o)$. Let $\mathcal{T}'$ be a definitional tree of the root of $\tau(t|_o)$. $\tau_{|V}$ is a constructor substitution since the patterns of definitional trees are linear patterns. Thus, $\tau(t|_o)$ contains fewer defined operation symbols than t. Therefore, if $t|_o$ is a ground subterm, by Proposition 2, there are no distinct triples in $\lambda(t, \mathcal{T})$ and the claim vacuously holds. Otherwise, $t|_o$ is not ground and by the induction hypothesis if (p_1, R_1, σ_1) and (p_2, R_2, σ_2) are distinct triples in $\lambda(\tau(t|_o), \mathcal{T}')$, then σ_1 and σ_2 are disjoint on $\mathcal{V}ar(\tau(t|_o))$. Let us show by contradiction that $\sigma_1 \circ \tau$ and $\sigma_2 \circ \tau$ are disjoint on V. Assume the opposite, i.e., there exist substitutions θ_1 and θ_2 such that $\theta_1 \circ \sigma_1 \circ \tau =_{\mathcal{R}} \theta_2 \circ \sigma_2 \circ \tau[V]$. Let v be any variable in $\mathcal{V}ar(\tau(t|_o))$. Then, there exists a variable $z \in \mathcal{V}ar(t|_o)$ such that $v \in \mathcal{V}ar(\tau(z))$ and $\theta_1 \circ \sigma_1 \circ \tau(z) =_{\mathcal{R}} \theta_2 \circ \sigma_2 \circ \tau(z)$. Since $\tau_{|V}$ is a constructor substitution, $\tau(z)$ is a constructor term. Thus, Proposition 5 implies that $\theta_1 \circ \sigma_1(v) =_{\mathcal{R}} \theta_2 \circ \sigma_2(v)$. Consequently, σ_1 and σ_2 are not disjoint on $\mathcal{V}ar(\tau(t|_o))$ contrary to the induction hypothesis. Thus, $\sigma_1 \circ \tau$ and $\sigma_2 \circ \tau$ are disjoint on V. Hence, the claim holds. □

The next theorem claims that no redundant derivation is computed by λ.

Theorem 5 (Disjointness of solutions) *Let $\mathcal{R}$ be an inductively sequential rewrite system extended by the equality rules, e an equation to solve and $V = \mathcal{V}ar(e)$. Let $e \overset{+}{\rightsquigarrow}_{\sigma}$ true and $e \overset{+}{\rightsquigarrow}_{\sigma'}$ true be two distinct derivations computed by λ. Then, σ and σ' are disjoint on V.*

Proof First, we prove the claim when the initial steps of $e \overset{+}{\rightsquigarrow}_{\sigma}$ *true* and $e \overset{+}{\rightsquigarrow}_{\sigma'}$ *true* differ. By our assumption, the derivations that we are considering are of the forms $e \rightsquigarrow_{\sigma_1} e_1 \overset{*}{\rightsquigarrow}_{\sigma_2}$ *true* and $e \rightsquigarrow_{\sigma_3} e_1' \overset{*}{\rightsquigarrow}_{\sigma_4}$ *true*. This implies that σ_1 and σ_3 belong to distinct triples in $\lambda(e, \mathcal{T})$, where $\mathcal{T}$ is a definitional tree of $\approx$. Notice that, by Proposition 2, equation e is not ground. By Proposition 8, the substitutions σ_1 and σ_3 are disjoint on V. By definition of disjoint substitutions, for all substitutions θ_1 and θ_3, $\theta_1 \circ \sigma_1 \neq_{\mathcal{R}} \theta_3 \circ \sigma_3[V]$. Since $\sigma = \sigma_2 \circ \sigma_1$ and $\sigma' = \sigma_4 \circ \sigma_3$, we deduce that σ and σ' are disjoint on V.
Now, we consider the general case. By our assumption, the derivations that we are considering are of the forms $e \overset{+}{\rightsquigarrow}_{\sigma_1} e_i \overset{+}{\rightsquigarrow}_{\sigma_2}$ *true* and $e \overset{+}{\rightsquigarrow}_{\sigma_1} e_i \overset{+}{\rightsquigarrow}_{\sigma_3}$ *true*, for some $i > 1$. The sub-derivations computed by λ, $e_i \overset{+}{\rightsquigarrow}_{\sigma_2}$ *true* and $e_i \overset{+}{\rightsquigarrow}_{\sigma_3}$ *true*, start from the same equation e_i and their initial steps differ. Thus, by Proposition 2, equation e_i is not ground. We have proved that in this case σ_2 and σ_3 are disjoint on $\mathcal{V}ar(e_i)$. Hence, by definition of disjoint substitutions we have

$$\theta_2 \circ \sigma_2 \neq_{\mathcal{R}} \theta_3 \circ \sigma_3[\mathcal{V}ar(e_i)] \quad \text{for all substitutions } \theta_2 \text{ and } \theta_3. \tag{3}$$

We prove by contradiction that the substitutions $\sigma_2 \circ \sigma_1$ and $\sigma_3 \circ \sigma_1$ are disjoint on V. So, we assume the opposite, i.e., there exist two substitutions β_2 and β_3 with

$$\beta_2 \circ \sigma_2 \circ \sigma_1 =_{\mathcal{R}} \beta_3 \circ \sigma_3 \circ \sigma_1[V] \tag{4}$$

Let y be a variable in $\mathcal{V}ar(e_i)$. By Proposition 7, we know that $\exists\, z \in V$ such that $y \in \mathcal{V}ar(\sigma_1(z))$. Since $\sigma_1(z)$ is a constructor term (by Proposition 6) and $y \in \mathcal{V}ar(\sigma_1(z))$, from Proposition 5, we deduce that

$$\beta_2 \circ \sigma_2(y) =_{\mathcal{R}} \beta_3 \circ \sigma_3(y) \tag{5}$$

and consequently that σ_2 and σ_3 are not disjoint on $\mathcal{V}ar(e_i)$, which contradicts (3). Thus the assumption (4) is false. Hence, the claim holds. □

We now discuss the cost and length of a derivation computed by our strategy. Our results are for the most part an extension of similar results for rewriting. We begin by extending to narrowing the notions of narrowing *multistep*, *family* of redexes, and *complete* step.

If p is a needed position of some term t, then in any narrowing derivation of t to a constructor term there is at least one step associated with p. If this step is delayed and p is not outermost, then several descendants of p may be created and several steps may become necessary to narrow this set of descendants (see Example 5). However, from a practical standpoint, if terms are appropriately represented, the cost of narrowing t at (some descendant of) p is largely independent of where the step occurs in the derivation of t. We formalize this viewpoint, which leads to another optimality result for our strategy.

It is well known [41] that, under appropriate conditions, a set of redexes can be reduced simultaneously. This notion can be extended to narrowing steps.

Definition 16 Let $t \leadsto_{p^i, l^i \to r^i, \sigma^i} t^i$, for i in some set of *indices* $I = \{1, \ldots, n\}$, be a narrowing step such that for any distinct i and j in I, p^i and p^j are disjoint and $\sigma^i \circ \sigma^j = \sigma^j \circ \sigma^i$. We say that t is narrowable to t' in a *multistep*, denoted $t \leadsto_{\langle p^i, l^i \to r^i, \sigma^i \rangle_{i \in I}} t'$, iff $t' = \circ_{\,i \in I}\, \sigma^i(((t[r^1]_{p^1})[r^2]_{p^2}) \ldots [r^n]_{p^n})$, where $\circ_{\,i \in I}\, \sigma^i$ denotes the composition $\sigma^n \circ \ldots \circ \sigma^2 \circ \sigma^1$ (the order is irrelevant).

When we want to emphasize the difference between a step as defined in Definition 6 and a multistep, we refer to the former as *elementary.* Otherwise, we identify an elementary step with a multistep in which the set of narrowed positions has just one element. A narrowing multistep can be thought of as a set of elementary steps performed in parallel. In fact, the conditions that we impose on the positions and substitutions of each elementary step from which a multistep is defined imply that, in a multistep, the order in which substitutions are composed and positions are narrowed is irrelevant.

The notion of multistep is essential for defining the cost of a derivation. As expected, the cost of a derivation is the total cost of its steps and an elementary step has unit cost. However, it does not seem appropriate, for practical reasons, to set the cost of a multistep equal to the number of positions narrowed in the step. A justification of this choice will be given after the definition of *cost.* The notions of *family* of redexes, *cost* of a derivation, and *complete* narrowing step defined next extend those for rewriting [9, 41, 50].

For any set I and equivalence relation $\sim$ on I, $|I|$ denotes the cardinality of I, and $I/\!\sim$ denotes the quotient of I modulo $\sim$.

Definition 17 Let $\alpha = t_0 \leadsto_{\langle p_1^i, R_1^i, \sigma_1^i \rangle_{i \in I_1}} t_1 \leadsto_{\langle p_2^i, R_2^i, \sigma_2^i \rangle_{i \in I_2}} \cdots$ be a narrowing (multi)derivation. Let σ_n be a shorthand for $\circ_{\,k \in I_n} \sigma_n^k$. The symbol $\sim_n$ denotes the equivalence relation on I_n defined as follows: for any i and j in I_n, $i \sim_n j$ iff the subterms identified by these indices have a common ancestor, more precisely, there exists some m, less than n, such that for some position q in t_m, both $t_n|_{p_{n+1}^i}$ and $t_n|_{p_{n+1}^j}$ are descendants of $\sigma_n \circ \sigma_{n-1} \cdots \sigma_{m+1}(t_m|_q)$.

We call a *family* of I_n any set of $\sim_n$-equivalent indices, and a *family* of t_n any set of redexes whose corresponding indices are $\sim_n$-equivalent.

The *cost* of the n-th step of α is the number of families in I_n, i.e., $|I_n/{\sim_n}|$. The *cost* of α, denoted $cost(\alpha)$, is the total cost of the steps of α.

We say that a family is *complete* iff it cannot be enlarged, and we say that a step is *complete* iff it contracts only complete families, more precisely, I_n is *complete* iff if i is in I_n, then for any position q of t_{n-1} such that p_n^i and q have a common ancestor in some term of α, there exists some j in I_n such that $q = p_n^j$. We say that a derivation is *complete* iff all its steps are complete.

We say that a narrowing multistep $t \rightsquigarrow_{\langle p^i, R^i, \sigma^i \rangle_{i \in I}} u$ is *needed* (respectively *outermost-needed*) iff each family of I contains a needed (respectively outermost-needed) narrowing position. A derivation is *needed* (respectively *outermost-needed*) iff all its steps are needed (respectively outermost-needed).

Example 11 Consider the rule for *double* in Example 5. Then

$$\begin{array}{lll} double(X+X) & \rightsquigarrow_{\langle \Lambda, \mathrm{R}_6, \{\} \rangle} & (X+X)+(X+X) \\ & \rightsquigarrow_{\langle 1, \mathrm{R}_4, \{X \mapsto 0\} \rangle \langle 2, \mathrm{R}_4, \{X \mapsto 0\} \rangle} & 0+0 \\ & \rightsquigarrow_{\langle \Lambda, \mathrm{R}_4, \{\} \rangle} & 0 \end{array}$$

is a narrowing multiderivation where all steps are complete and have cost 1.

If I is the set of indices of a narrowing step and i and j belong to I, then $i \sim j$ iff p_i and p_j are, using an anthropomorphic metaphor, blood related. A complete derivation is characterized by narrowing complete "families," i.e., sets containing all the pairwise blood related subterms of a term. Note that the blood related subterms of a term are all equal and that their positions are pairwise disjoint; thus, all of them can be included in a multistep. Our choice of cost measure is suggested by the observation that if $t \rightsquigarrow_{p,R,\sigma} t'$ and p and q are blood related positions, then narrowing t at q "when t is being narrowed at p" involves no additional computation of a substitution and/or a rule and, consequently, no additional computation of a substituting term (the instantiation of the right-hand side of a rule) since the reducts of blood related subterms are all equal, too.

We show that complete, outermost-needed narrowing derivations have minimum cost and minimum length. The proof relies on the analogous result for orthogonal systems formulated for reduction sequences only [41, 50]. Formally, we must give a meaning to the notion of *need* when a non-elementary step is computed. To achieve optimality, we require multisteps only as far as blood related terms are concerned. Thus, it suffices to consider multisteps in which only one complete family is narrowed. These steps are quite similar to elementary steps when, in the representation of a term, blood related subterms are fully "shared."

The framework of term graph rewriting [68] offers a formal model that leads to a simple and efficient implementation of our strategy. We consider only finite term and acyclic graphs. The computation of a complete needed narrowing step in a term graph, g, proceeds as follows: we *unravel* g, thus obtaining a "regular" term t; we compute an outermost-needed narrowing step of t using λ as described earlier; we map back the computed step to g; and finally we perform the narrowing step on g. The adequacy of this approach for the rewrite systems and the terms that we are considering is discussed in [7, 46]. In practice, the computation of a step is simple and efficient. Since λ computes a narrowing step by unification, the unraveling of a term graph g is achieved simply by traversing the representation of g as if it were a tree during the unification phase, and no map-back operation is actually required. In the following, we use the notion of *reduction multisteps*—narrowing multisteps where the step's unification is limited to a matching (cf. Definition 17).

Definition 18 Let $\alpha = t_0 \rightsquigarrow_{\langle p_1^i, R_1^i, \sigma_1^i \rangle_{i \in I_1}} t_1 \rightsquigarrow_{\langle p_2^i, R_2^i, \sigma_2^i \rangle_{i \in I_2}} \cdots \rightsquigarrow_{\langle p_q^i, R_q^i, \sigma_q^i \rangle_{i \in I_q}} t_q$ be a narrowing multiderivation. Let $\theta_j = \circ_{k \in I_q} \sigma_q^k \circ \cdots \circ_{k \in I_j} \sigma_j^k$. We say that $A = \theta_1(t_0) \xrightarrow{*} t_q$ is the reduction sequence *canonically associated* to α iff for every $n > 0$, if $t_{n-1} \rightsquigarrow_{\langle p_n^i, R_n^i, \sigma_n^i \rangle_{i \in I_n}} t_n$ is the n-th step of α, then $\theta_n(t_{n-1}) \rightarrow_{\langle p_n^i, R_n^i \rangle_{i \in I_n}} \theta_{n+1}(t_n)$ is the n-th step of A.

Loosely speaking, α and A compute the same steps.

Lemma 6 *Let $\mathcal{R}$ be an inductively sequential rewrite system, $\alpha = t_0 \rightsquigarrow_{\langle p_1^i, R_1^i, \sigma_1^i \rangle_{i \in I_1}} t_1 \rightsquigarrow_{\langle p_2^i, R_2^i, \sigma_2^i \rangle_{i \in I_2}} \cdots \rightsquigarrow_{\langle p_q^i, R_q^i, \sigma_q^i \rangle_{i \in I_q}} t_q$ a narrowing multiderivation and A the reduction sequence canonically associated to α. If α is needed, then A is needed, too.*

Proof Suppose that α is not null and that $t_{n-1} \rightsquigarrow_{\langle p_n^i, R_n^i, \sigma_n^i \rangle_{i \in I_n}} t_n$ is the n-th step of α, for some $n > 0$. The n-th step of A is $\theta_n(t_{n-1}) \rightarrow_{\langle p_n^i, R_n^i \rangle_{i \in I_n}} \theta_{n+1}(t_n)$. For each family in I_n, there exists an index l such that the narrowing position p_n^l is needed. Since $\circ_{k \in I_n} \sigma_n^k \leq \theta_n$, by Definition 10, the position p_n^l is also needed in A. □

Lemma 7 *Let $\mathcal{R}$ be an inductively sequential rewrite system, $\alpha = t_0 \rightsquigarrow_{\langle p_1^i, R_1^i, \sigma_1^i \rangle_{i \in I_1}} t_1 \rightsquigarrow_{\langle p_2^i, R_2^i, \sigma_2^i \rangle_{i \in I_2}} \cdots \rightsquigarrow_{\langle p_q^i, R_q^i, \sigma_q^i \rangle_{i \in I_q}} t_q$ and A the reduction sequence canonically associated to α. If α is complete, then A is also complete.*

Proof We prove that if A is incomplete, then so is α. Suppose that α is not null and that $t_{n-1} \rightsquigarrow_{\langle p_n^i, R_n^i, \sigma_n^i \rangle_{i \in I_n}} t_n$, for some $n > 0$, is the first step of α such that the corresponding step of A, $\theta_n(t_{n-1}) \rightarrow_{\langle p_n^i, R_n^i \rangle_{i \in I_n}} \theta_{n+1}(t_n)$, is incomplete. There exists a position q of t_{n-1} such that for no $i \in I_n$, $q = p_n^i$, and for some $j \in I_n$, q and p_n^j are blood related. In other words, the family of I_n to which p_n^j belongs is not complete. Since q and p_n^j are blood related, they have a common ancestor. Since the first incomplete step occurs at t_{n-1}, the common ancestor occurs in some preceding term t_l with $l \leq n-2$. Hence, there exists a position r in $\theta_{l+1}(t_l)$ such that both q and p_n^j are (rewriting) descendants of r. Since the θ_i's are constructor substitutions and the root of $t_l|_r$ is a defined operation, r is a position of t_l as well. Thus, both q and p_n^j are (narrowing) descendants of r in t_{n-1} and the n-th step of α is also incomplete. □

Corollary 1 *If $\alpha = t \overset{*}{\rightsquigarrow}_\sigma u$ is a complete and needed narrowing multiderivation of a term t into a constructor term u, then α has minimum cost. I.e., for any multiderivation $\beta = t \overset{*}{\rightsquigarrow}_{\sigma'} u$ with $\sigma = \sigma'[\mathcal{V}ar(t)]$, $cost(\alpha) \leq cost(\beta)$.*

Proof Let A and B be the reduction sequences canonically associated with the narrowing derivations α and β, respectively. By Definition 18, $\sigma = \sigma'[\mathcal{V}ar(t)]$ implies that the initial and final terms of A and B are pairwise identical. By Lemmas 6 and 7, A is a complete and needed reduction sequence. Thus, $cost(A) \leq cost(B)$ [41, 50]. By Definition 17, $cost(A) = cost(\alpha)$ and $cost(B) = cost(\beta)$. Thus, α has minimum cost among all the narrowing derivations computing $t \overset{*}{\rightsquigarrow}_\sigma u$. □

Completeness is essential to achieve minimum cost. In fact, if we stick to elementary derivations, the outermost-needed strategy yields the *longest* derivation among those that narrow terms only at needed positions. Non-trivial families of blood related subterms are created only by non-right-linear rules. The following example highlights these issues:

Example 12 We follow up on Example 5. It is immediate to verify that an outermost-needed narrowing elementary derivation (actually, a reduction sequence since the term is ground) of $double(0+0)$ has 4 steps. The following elementary derivation is shorter.

$$double(0+0) \leadsto_{1,\mathrm{R}_4,\{\}} double(0) \leadsto_{\Lambda,\mathrm{R}_6,\{\}} 0+0 \leadsto_{\Lambda,\mathrm{R}_4,\{\}} 0$$

This derivation is shorter than the outermost-needed one, because its first step narrows the subterm at position 1. By contrast, the first step of the outermost-needed derivation narrows the initial term at the root. This step yields two descendants of position 1, which are both needed. Sharing these blood related subterms would save one step.

6 Extensions of needed narrowing

Since the preliminary publication of needed narrowing in [5], both needed narrowing and a combination of other strategies originating from this idea have become the preferred choice for the evaluation of modern narrowing-based functional logic languages. We will come back later to the reasons of this acceptance. In this section, we discuss variations and extensions of needed narrowing to classes of functional logic programs which are more general than the inductively sequential ones.

The key idea behind needed narrowing is the discovery that certain steps of a computation (referred to as "needed") must be performed to obtain the computation's result. Definitional trees allow us to efficiently compute needed steps. For instance, [32] describes an efficient implementation of needed narrowing by a straightforward transformation of definitional trees into Prolog programs, and [70] presents an abstract machine suitable for the direct compilation of needed narrowing into machine code. However, definitional trees also limit the domain of needed narrowing to inductively sequential systems. Inductive sequentiality excludes systems with overlapping left-hand sides, such as the following one, that are interesting in functional logic programming.

Example 13 The following definition of Boolean disjunction is known as the *parallel-or*:

$$\begin{array}{rcll} X \vee true & \to & true & \mathrm{R}_1 \\ true \vee X & \to & true & \mathrm{R}_2 \\ false \vee false & \to & false & \mathrm{R}_3 \end{array}$$

This system is not inductively sequential since there is no definitional tree of "$\vee$." This can be inferred by the fact that neither argument of "$\vee$" is demanded by all the rules. The first two rules of "$\vee$" are overlapping. Both rules are applicable, e.g., to the term $true \vee true$. As a consequence, a term of the form $t_1 \vee t_2$ may be narrowed to normal form by narrowing either t_1 or t_2, although it is not known how to make this choice without looking ahead. A further consequence is the existence of terms of the form $t_1 \vee t_2$ in which neither t_1 nor t_2 need to be evaluated—condition contrary to the key idea of needed narrowing.

The rules of parallel-or belong to the class of the weakly orthogonal, constructor-based rewrite systems which allow overlapping among the rules' left-hand sides as long as any critical pair is trivial [15, 47]. A variation of needed narrowing, applicable to this class of rewrite systems, is obtained by extending definitional trees with a new kind of nodes providing a mechanism for the non-deterministic selection of arguments to be evaluated [2, 6]. These definitional trees, called *parallel*, have the additional form $or(\mathcal{T}_1, \ldots, \mathcal{T}_k)$, where $k > 1$ and all subtrees $\mathcal{T}_j$ have the same pattern, but different arguments in the immediately subsequent branch nodes. The parallel definitional tree of the operation of Example 13 is

$$\begin{array}{l} or(branch(X_1 \vee X_2, 1, leaf(true \vee X_2 \to true), \ldots), \\ \quad branch(X_1 \vee X_2, 2, leaf(X_1 \vee true \to true), \ldots)) \end{array}$$

Weakly needed narrowing [6] is an extension of needed narrowing to weakly orthogonal, constructor-based rewrite systems. This strategy is computed by a function similar to λ that non-deterministically selects a subtree of each *or* node that is encountered during the computation of a narrowing step. This strategy is almost identical to the demand driven narrowing strategy proposed in [49] (which is presented without proofs of soundness and completeness) and is a conservative extension of needed narrowing. Since weakly orthogonal term rewriting systems lack a notion of needed redex, the strong optimality results of needed narrowing cannot be preserved by weakly needed narrowing.

By contrast to needed narrowing, weakly needed narrowing, as well as any other previously proposed narrowing strategy for weakly orthogonal systems, does not evaluate every ground term in a deterministic way. *Parallel narrowing* [6] is a variation of weakly needed narrowing that deterministically evaluates every ground term. It achieves this property by narrowing in a single step several distinct subterms similarly to the rewriting strategy of Sekar and Ramakrishnan [66]. Parallel narrowing is better than weakly needed at pruning the search space. For instance, consider Example 13 together with the additional rule

$$f(a) \;\rightarrow\; \mathit{true} \qquad \mathrm{R}_4$$

and the term $f(X) \vee f(X)$. Weakly needed narrowing computes four distinct derivations all with the same substitution, $\{X \mapsto a\}$.

$$\begin{array}{lllllll}
f(X) \vee f(X) & \rightsquigarrow_{1,\mathrm{R}_4,\{X\mapsto a\}} & \mathit{true} \vee f(a) & \rightsquigarrow_{\Lambda,\mathrm{R}_2,id} & \mathit{true} & & \\
f(X) \vee f(X) & \rightsquigarrow_{1,\mathrm{R}_4,\{X\mapsto a\}} & \mathit{true} \vee f(a) & \rightsquigarrow_{2,\mathrm{R}_4,id} & \mathit{true} \vee \mathit{true} & \rightsquigarrow_{\Lambda,\mathrm{R}_2,id} & \mathit{true} \\
f(X) \vee f(X) & \rightsquigarrow_{2,\mathrm{R}_4,\{X\mapsto a\}} & f(a) \vee \mathit{true} & \rightsquigarrow_{\Lambda,\mathrm{R}_1,id} & \mathit{true} & & \\
f(X) \vee f(X) & \rightsquigarrow_{2,\mathrm{R}_4,\{X\mapsto a\}} & f(a) \vee \mathit{true} & \rightsquigarrow_{1,\mathrm{R}_4,id} & \mathit{true} \vee \mathit{true} & \rightsquigarrow_{\Lambda,\mathrm{R}_2,id} & \mathit{true}
\end{array}$$

Parallel narrowing computes the same substitution and simultaneously reduces both arguments of "$\vee$", yielding the unique parallel narrowing derivation

$$f(X) \vee f(X) \;\; \approx\!\!\!>_{\{X\mapsto a\}} \;\; \mathit{true} \vee \mathit{true} \;\; \approx\!\!\!>_{id} \;\; \mathit{true}$$

Parallel narrowing is a conservative extension of two optimal evaluation strategies. It behaves as needed narrowing on inductively sequential rewrite systems, and is identical to Sekar and Ramakrishnan's rewrite strategy [66] on ground terms. Parallel narrowing is the only complete strategy for functional logic programs that evaluates ground terms in a fully deterministic way—even in the presence of overlapping and non-terminating rules.

A second extension of needed narrowing deals with so-called *non-determinate functions*. The *choice* operator is a perfect example of this idea. Non-determinate computations are modeled by overlapping rewrite systems, e.g.,

$$\begin{array}{lll}
\mathit{choice}(X,Y) & \rightarrow & X \\
\mathit{choice}(X,Y) & \rightarrow & Y
\end{array}$$

Non-determinate functions support a terse and elegant programming style. For example, the following program non-deterministically computes a permutation of a list.

$$\begin{array}{rcl}
\mathit{insert}(X,Xs) & \rightarrow & \mathit{choice}([X|Xs], \mathit{insert}_1(X,Xs)) \\
\mathit{insert}_1(X,[Y|Ys]) & \rightarrow & [Y|\mathit{insert}(X,Ys)] \\
\mathit{permute}([]) & \rightarrow & [] \\
\mathit{permute}([X|Xs]) & \rightarrow & \mathit{insert}(X, \mathit{permute}(Xs))
\end{array}$$

The use of non-determinate functions in functional logic programming has been investigated by [1, 25]. In particular, Gonzales Moreno et al. [25] present a sound and complete narrowing calculus for a very large class of overlapping constructor-based systems which allow non-determinate computations. A more sophisticated narrowing strategy for a more limited class of overlapping systems is given in [4] along the lines of needed narrowing. Overlapping rules in these systems have the same left-hand side except for a renaming of variables. Rules with overlapping left-hand sides are merged by introducing a "choice" in the right-hand side (as shown by the rule of *insert* in the previous program). Definitional trees in these systems are exactly the definitional trees discussed in this paper—except for *leaf* nodes which may contain several choices. Systems in which every function has one such tree are called *overlapping* inductively sequential. *Inductively sequential narrowing* [4] is a strategy for possibly overlapping inductively sequential systems. This strategy is simply obtained from needed narrowing by adding, where appropriate, a choice of a right-hand side. This strategy preserves, with a few exceptions, the most important properties of needed narrowing including a weaker form of optimality modulo non-determinate choices.

A third extension of needed narrowing deals with *higher-order programming*; a feature that greatly contributes to the power of functional languages and has been investigated in the context of functional logic languages by [35, 57]. As shown by Warren [72] for the case of logic programming, one can extend a first-order language to cover the higher-order features of existing functional languages (including partial applications and lambda abstractions) by providing an application function $apply(F, X) \to F(X)$. This function can be explicitly defined by first-order rules through an encoding of function names as constructors. By this method, we extend needed narrowing to higher-order functional programming since the rules defining the *apply* function are inductively sequential. More expressive languages should allow lambda abstractions in patterns (i.e., as arguments in the left-hand sides of rules)—a use that goes beyond current higher-order functional languages. For instance, the following higher-order rules for symbolic differentiation cannot be handled by existing functional languages, but are plausible in a higher-order functional logic language:

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

A higher-order functional logic language is able to synthesize new functions satisfying a given goal. For instance, the equation

$$\lambda x.\mathit{diff}(\lambda y.\mathit{sin}(F(x, y))), x) \approx \lambda x.\mathit{cos}(x)$$

is solved by instantiating the functional variable F by the projection function $\lambda x.\lambda y.y$. Function synthesis is a complex task that requires higher-order unification. A strategy for higher-order narrowing, based on definitional trees, has been proposed in [35]. This strategy computes a minimal set of (higher-order) solutions by extending needed narrowing to deal with lambda abstractions.

A fourth extension of needed narrowing deals with graph rewriting systems [19]. Future functional and logic programming languages will handle graph structures explicitly, as is yet the case in the declarative language Clean [63]. There are many reasons which motivate the use of graphs. They actually allow sharing of subexpressions, which leads to efficient implementations. They also permit to go beyond the processing of first-order terms by efficiently handling real-world data structures (e.g., data bases). In [19], a class of confluent constructor-based graph rewriting systems has been proposed for which needed narrowing has been generalized successfully with the same nice optimality results.

Finally, the ideas of (weakly) needed narrowing and definitional trees have also been extended by concurrency features. This approach provides a unified computation model which encompasses the most important existing declarative programming paradigms [33]. This extension is the foundation of Curry [33, 37], a new multi-paradigm programming language aiming to integrate functional, logic, and concurrent programming paradigms. The operational model of Curry uses a slightly extended form of definitional trees to specify the evaluation strategy of each function.

As the field of functional logic programming evolves, more demands are placed on narrowing strategies. In addition to soundness, completeness, and efficiency, a strategy must cope with results, features and techniques adopted in this field. One such feature is *conditional* rules that contribute to the expressiveness of programs. For instance, the following rules define a predicate for list membership on the basis of the function *append* to concatenate lists:

$$\begin{array}{rcl} \mathit{append}([\,],L) & \to & L \\ \mathit{append}([E|R],L) & \to & [E|\mathit{append}(R,L)] \\ \mathit{member}(E,L) & \to & \mathit{true} \;\Leftarrow\; \mathit{append}(L1,[E|L2]) \approx L \end{array}$$

Variables $L1$ and $L2$ do not occur in the left-hand side of their rule; hence, they don't get a value when the rule is fired. However, narrowing is an ideal computation to instantiate them. For example, in order to reduce a term like $\mathit{member}(b,[a,b,c])$, we can compute (e.g., by narrowing) values for the extra variables $L1$ and $L2$ so that the condition "$\mathit{append}(L1,[b|L2]) \approx [a,b,c]$" is reducible to *true*. If the left-hand sides of rewrite rules are non-overlapping, one can easily translate conditional rules into unconditional ones [8]. This method has also been used in the functional logic language BABEL [55] to define the semantics of conditional rules. In our example, we transform the rule for *member* into an unconditional one by interpreting the condition and the right-hand side as a "conditional expression":

$$\begin{array}{rcl} \mathit{member}(E,L) & \to & \mathit{cond}(\mathit{append}(L1,[E|L2]) \approx L, \mathit{true}) \\ \mathit{cond}(\mathit{true},X) & \to & X \end{array}$$

On an inductively sequential source system, this transformation yields an inductively sequential target system. Hence we can apply needed narrowing to the target system, although not all the properties of needed narrowing may be preserved by the transformation.

7 Comparison with other narrowing strategies

The trade-off between power and efficiency is central to the use of narrowing, especially in programming. To this aim, several narrowing strategies, e.g., [11, 14, 17, 18, 21, 22, 23, 20, 28, 38, 39, 43, 51, 52, 53, 54, 55, 59, 65, 73] have been proposed. The notion of completeness has evolved accordingly. Plotkin's classic formulation [64] has been relaxed to completeness w.r.t. ground solutions (e.g. [22]) or completeness w.r.t. *strict* equality and domain-based interpretations, as in [23, 55]. The latter appear more appropriate for narrowing as the computational paradigm of functional logic programming languages in the presence of infinite data structures and computations. For this purpose, we have introduced the equality rules in Section 2 and defined the computation of solutions of an equation as narrowing the equation to *true*. Although this approach is also taken in other functional logic languages, e.g., *K-LEAF* [23] or *BABEL* [55], it has the disadvantage that, in some cases, it simply enumerates all constructor terms.

Example 14 Consider the signature of Example 2 and the equation $s(A) \approx s(s(B))$. All possible solutions of this equation can be described by $\sigma \circ \{A \mapsto s(B)\}$ where σ is an arbitrary ground constructor substitution with $B \in \mathcal{D}om(\sigma)$. However, applying needed narrowing to this equation w.r.t. the equality rules produces an infinite set of derivations with outcomes $\{A \mapsto s(0), B \mapsto 0\}$, $\{A \mapsto s(s(0)), B \mapsto s(0)\}$, $\{A \mapsto s(s(s(0))), B \mapsto s(s(0))\}$, etc.

In order to avoid such trivial but infinite search spaces, we can replace narrowing of constructor terms by a computation of the most general unifier. If $t_1 \approx t_2$ is an equation with $t_1, t_2 \in \mathcal{T}(\mathcal{C}, \mathcal{X})$, we compute the most general unifier of t_1 and t_2 instead of applying narrowing steps. If both terms are not unifiable, there is no solution of $t_1 \approx t_2$. Otherwise, $mgu(t_1, t_2)$ represents all solutions of $t_1 \approx t_2$ in the sense that for each solution σ there exists a ground constructor substitution τ with $\sigma = \tau \circ mgu(t_1, t_2)$. Note that the restriction to constructor substitutions is essential since $\{A \mapsto s(f(0)), B \mapsto f(0)\}$ is not a solution of the equation $s(A) \approx s(s(B))$ if the operation f is defined by

$$f(0) \to f(0) \quad .$$

The computation of the most general unifier in the final step of a narrowing derivation is also done in other narrowing strategies for terminating rewrite systems (e.g., [14, 17, 22]).

We briefly recall the underlying ideas of a few major strategies and compare them with ours using the following example. We choose a terminating rewrite system with completely defined operations, otherwise all the eager strategies would be immediately excluded.

Example 15 The symbols a, b, and c are constructors, whereas f and g are defined operations.

$$\begin{array}{rcll}
f(a) & \to & a & \mathrm{R}_1 \\
f(b(X)) & \to & b(f(X)) & \mathrm{R}_2 \\
f(c(X)) & \to & a & \mathrm{R}_3
\end{array}
\qquad
\begin{array}{rcll}
g(a, X) & \to & b(a) & \mathrm{R}_4 \\
g(b(X), a) & \to & a & \mathrm{R}_5 \\
g(b(X), b(Y)) & \to & c(a) & \mathrm{R}_6 \\
g(b(X), c(Y)) & \to & b(a) & \mathrm{R}_7 \\
g(c(X), Y) & \to & b(a) & \mathrm{R}_8
\end{array}$$

The equation to solve is $g(X, f(X)) \approx c(a)$. Our strategy computes only three derivations—only one of which yields a solution.

$$\begin{array}{l}
g(X, f(X)) \approx c(a) \ \rightsquigarrow_{1, \mathrm{R}_4, \{X \mapsto a\}} \ b(a) \approx c(a) \\
g(X, f(X)) \approx c(a) \ \rightsquigarrow_{1, \mathrm{R}_8, \{X \mapsto c(X_1)\}} \ b(a) \approx c(a) \\
g(X, f(X)) \approx c(a) \ \rightsquigarrow_{1.2, \mathrm{R}_2, \{X \mapsto b(X_1)\}} \ g(b(X_1), b(f(X_1))) \approx c(a) \ \overset{*}{\rightsquigarrow}_{\{\}} \ true
\end{array}$$

Basic narrowing [42] avoids positions introduced by the instantiations of previous steps. Its completeness, and that of its variations, e.g., [11, 38, 39, 51, 59, 58], is known for confluent and terminating rewrite systems (see [51] for a systematic study.) This strategy may perform useless steps and computes an infinite search space for our benchmark example. LSE narrowing [11], a refinement of basic narrowing with additional redundancy tests, ensures that the same solution (up to variable renaming) is not computed by two different derivations. However, it may be the case that one solution is an instance of another. Hence different solutions computed by LSE narrowing are not incomparable in general.

Innermost narrowing [22] narrows only patterns. It is ground complete only for terminating constructor-based systems with completely defined operations. It may perform useless steps and it computes an infinite number of derivations for our benchmark example.

Outermost narrowing [17, 18, 20] narrows outermost operation-rooted terms. This strategy is ground complete only for a restrictive class of rewrite systems. It computes no solution for our benchmark example unless we transform the considered rewrite system [17].

Lazy narrowing [23, 28, 54, 55, 65] narrows an inner term only when the step is demanded to narrow an outer term. For these strategies, the qualifier "lazy" is used as a synonym of "outermost" or "demand driven," rather than in the technical sense we propose. The completeness of these strategies is generally expensive to achieve: [28] requires an ad-hoc implementation of backtracking, with the potential of evaluating some term several times; [23] requires flattening of functional nesting and a specialized WAM-like machine in which terms are dynamically reordered; and [54] requires a transformation of the rewrite system which, for our benchmark example, increases the number of operations and lengthen the derivations. The following example shows a further important difference between lazy narrowing and needed narrowing:

Example 16 Consider the following term rewriting system:

$$\begin{array}{rcll} \mathit{one}(0) & \to & s(0) & \mathrm{R}_1 \\ \mathit{one}(s(X)) & \to & \mathit{one}(X) & \mathrm{R}_2 \end{array} \qquad \begin{array}{rcll} f(0,0) & \to & 0 & \mathrm{R}_3 \\ f(s(X),0) & \to & s(0) & \mathrm{R}_4 \\ f(X,s(Y)) & \to & s(s(0)) & \mathrm{R}_5 \end{array}$$

A lazy narrowing strategy reduces a term to a head normal form before a narrowing rule is applied, when this reduction is required for the rule's application [54, 65]. For instance, if we wish to apply rule R_3 to the term $f(\mathit{one}(X), X)$, then f's argument $\mathit{one}(X)$ must be narrowed to head normal form in an attempt to unify the result with f's argument 0 in R_3. Unfortunately, there are infinitely many narrowing derivations of $\mathit{one}(X)$ to a head normal form term—for every $n \geq 0$, $\mathit{one}(X) \overset{*}{\leadsto}_{\{X \mapsto s^n(0)\}} s(0)$. In a sequential implementation of a lazy strategy, this would delay forever the application of rules R_4 and R_5, which would result in an incomplete implementation [28]. Apart from this behavior in sequential implementations, lazy narrowing may have an infinite search space which may be avoided by our strategy. For instance, there are infinitely many lazy narrowing derivations of the equation $f(\mathit{one}(X), X) \approx 0$ due to the infinite number of derivations of $\mathit{one}(X)$ to head normal form. But our strategy computes only the following two derivations:

$$\begin{array}{l} f(\mathit{one}(X),X) \approx 0 \ \leadsto_{1.1,\mathrm{R}_1,\{X\mapsto 0\}} \ f(s(0),0) \approx 0 \ \leadsto_{1,\mathrm{R}_4,\{\}} \ s(0) \approx 0 \\ f(\mathit{one}(X),X) \approx 0 \ \leadsto_{1,\mathrm{R}_5,\{X\mapsto s(Z)\}} \ s(s(0)) \approx 0 \end{array}$$

The good behavior of our strategy is due to the fact that we have changed the order of redex selection and variable instantiation: our strategy instantiates f's second argument X to one of the constructors 0 or s before the redex is selected. This avoids narrowing the uninstantiated argument $\mathit{one}(X)$.

Standard narrowing [14, 43, 52, 73], also called leftmost outside-in narrowing [43], extends to narrowing the notion of standard rewrite derivations introduced by Huet and Lévy [41]. Thus, standard narrowing characterizes narrowing derivations rather than narrowing steps. In these narrowing derivations, as in lazy narrowing, outer positions are considered before inner positions. This strategy is complete for orthogonal systems w.r.t. strict equality. Standard narrowing derivations are the same as needed narrowing derivations on the benchmark example. However, You gives no constructive procedure for enumerating standard narrowing derivations. Darlington and Guo sketch in [14] a narrowing strategy called lazy pattern driven narrowing which is intended to compute standard narrowing derivations for orthogonal constructor-based rewrite systems. This strategy is not optimal in the sense that it may generate several times a same standard derivation. On the benchmark example, this strategy generates fourteen standard derivations and three times the unique derivation that computes a solution for the equation $g(X, f(X)) \approx c(a)$. In [43], Ida and Nakahara give a procedure for enumerating standard narrowing derivations. They do not use the classic definition of a narrowing step (as in Definition 6), but break a narrowing step into more elementary inference steps (selection of a rule, pattern matching, etc). Due to the separation of rule application and pattern matching, their calculus has more non-deterministic choices than needed narrowing. For our benchmark example, their narrowing calculus non-deterministically applies five different inference steps to the initial equation. The same holds for other similar narrowing calculi, e.g., [27, 53]. Since this strategy is defined for orthogonal, but not necessarily constructor-based rewrite systems, it has also been extended [57] to provide a complete narrowing calculus for applicative term rewriting systems (which model the higher-order features of current functional languages). Recently, Middeldorp and Okui improved the strategies in [43, 53] by providing a deterministic lazy narrowing calculus [52] for orthogonal constructor-based rewrite systems (actually, they also provide completeness results for left-linear confluent constructor-based rewrite systems). This improved calculus is complete w.r.t. strict equality and behaves on the benchmark example as the narrowing calculus given in [43]. If we reconsider Example 16, all the quoted strategies including the last one, fail to stop on the equation $f(one(X), X) \approx 0$ contrary to needed narrowing. Nevertheless, standard narrowing has one optimality result due to the standardization of narrowing derivations. It has been shown in [52, 73] that the solutions computed by standard narrowing are always incomparable.

Implementations of narrowing in Prolog [3, 13, 44, 49] are proposed as a prototypical and portable integration of functional and logic languages. For example, [13, 44] have been proposed as an alternative to the specialized machines required for *K-LEAF* [23] and *BABEL* [55], respectively. The most recent proposals [3, 49] are based on definitional trees and appear to compute needed steps for inductively sequential systems, although both methods neither formalize nor claim this property. The scheme in [3] computes λ directly by unification. The patterns involved in the computation of λ are a superset of those represented by the leaves of a definitional tree. This is suggested by claim 1 of Theorem 1 that shows a "strong" need for the positions computed using λ—not only the terms at these positions must be eventually narrowed, but they must be eventually narrowed to *head normal forms*. The resulting implementation takes advantage of this characteristic and its performance appears to be superior to the other proposals. This property of needed narrowing is also emphasized by the benchmarks presented in [32] which show the superiority of needed narrowing in comparison to other narrowing strategies for Prolog-based implementations of functional logic programs. We

Strategy	Requirements				Completeness	Optimality		
	C	T	CB	others		IS	DS	length of derivations
basic [42]	X	X			complete	no	no	no
innermost [22]	X	X	X	completely defined operations	ground complete	(yes)		no
outermost [17, 18]	X	X	X	uniformity	ground complete	(yes)		no
lazy [23, 28, 54] [55, 65]	X		X	nonambiguity [55, p. 197]	complete w.r.t. strict equality	no	no	no
standard [14, 43] [52, 73]	X		(X)	orthogonality	complete w.r.t. strict equality	yes		
needed	X		X	inductive sequentiality	complete w.r.t. strict equality	yes	yes	yes

Figure 1: Summary of the characteristics of some major narrowing strategies.

summarize the characteristics of the major narrowing strategies in the table of Figure 1. In the requirement columns, *C*, *T* and *CB* denote "confluence," "termination," and "constructor discipline," respectively. In the optimality section, columns *IS* and *DS* denote incomparable and disjoint solutions, respectively. The entries "yes" and "no" mean that the corresponding property holds or does not hold, respectively. An empty entry indicates that the corresponding property has not been investigated in the literature. The entry "(yes)" means that the corresponding property has been shown under some additional conditions [17, Theorem 3].

It is interesting to note that lazy narrowing, as formally defined in [55], does not satisfy any of the optimality properties shown in the table. For instance, consider Example 2 and the goal $X \leq X + X$. Since both arguments of $\leq$ are demanded in the sense of [55], there are the following lazy narrowing derivations (among others):

$$\begin{array}{l} X \leq X + X \ \rightsquigarrow_{\Lambda, \mathrm{R}_1, \{X \mapsto 0\}} \ \mathit{true} \\ X \leq X + X \ \rightsquigarrow_{2, \mathrm{R}_4, \{X \mapsto 0\}} \ 0 \leq 0 \ \rightsquigarrow_{\Lambda, \mathrm{R}_1, \{\}} \ \mathit{true} \end{array}$$

Thus, the solution $\{X \mapsto 0\}$ is computed twice and the second derivation is not of minimal length.

To summarize, the distinguishing features of our strategy are the following: with respect to eager strategies, completeness for non-terminating rewrite systems; with respect to the so-called lazy strategies, a sharp characterization of laziness; with respect to any strategy, optimality and ease of computation.

8 Concluding remarks

We have proposed a new narrowing strategy obtained by extending to narrowing the well-known notion of *need* for rewriting. Need for narrowing appears harder to handle than need for rewriting—to compute a needed narrowing step one must also look ahead at a potentially infinite number of substitutions. Remarkably, there is an efficiently algorithm for this computation in inductively sequential systems.

The only non-trivial operation required for the implementation of our strategy is unification. Prolog interpreters and compilers offer efficient built-in implementations of unification. This is one reason for the good performance, as we have reported in the previous section, of implementations of needed narrowing in Prolog (see [32] for some benchmarks). In addition to being well-suited to efficient implementations of functional logic languages in Prolog, as described in [3, 32, 49], our strategy can be easily implemented in other languages. Unification has well-understood algorithms that are efficiently implemented in various programming styles and abstract machines. For instance, [36] describes the compilation of the functional logic language Curry (which is based on an extension of needed narrowing) into Java.

We have also shown that our strategy computes only disjoint and optimal derivations. Although all the previously proposed lazy strategies have the latter as their primary goal, our strategy is the only one for which this result is formalized and proved.

We want to conclude with a general assessment of the "overall quality" of the narrowing strategy used by a programming language. The key factor is the trade-off between the size of the class of rewrite systems for which the strategy is complete, and the efficiency of its computations.

We have proved both completeness *and* optimality for inductively sequential systems. We have also discussed how the idea of needed narrowing has been extended into different directions (weakly orthogonal systems, non-determinate functions, conditional rules, higher-order functions, graph rewrite systems). These extensions preserve some of the remarkable characteristics of needed narrowing, such as an inherent conceptual simplicity and efficiency of computations; compatibly with the larger domain in which they are employed; and the full preservation of the key feature of needed narrowing—the computation of needed steps—on the inductively sequential portions of a program.

Acknowledgement

The authors are grateful to the anonymous referees for their helpful remarks to improve the final version of this paper.

References

[1] S. Antoy. Non-determinism and lazy evaluation in logic programming. In T. P. Clement and K.-K. Lau, editors, *Proc. Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'91)*, pages 318–331, Manchester, UK, July 1991. Springer Workshops in Computing Series.

[2] S. Antoy. Definitional trees. In *Proc. International Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.

[3] S. Antoy. Needed narrowing in Prolog. In *Proc. Int. Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'96)*, pages 473–474. Springer LNCS 1140, 1996. Full version accessible at `http://www.cs.pdx.edu/~antoy`.

[4] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.

[5] 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.

[6] S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 138–152. MIT Press, 1997.

[7] H. Barendregt, M. van Eekelen, J. Glauert, R. Kennaway, M.J. Plasmeijer, and M. Sleep. Term graph rewriting. In *Proc. Parallel Architectures and Languages Europe (PARLE'87)*, pages 141–158. Springer LNCS 259, 1987.

[8] J.A. Bergstra and J.W. Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Sciences*, 32(3), 1986.

[9] G. Berry and J.-J. Lévy. Minimal and optimal computations of recursive programs. *Journal of the Association for Computing Machinery*, 26(1):148–175, 1979.

[10] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *Proc. European Symposium on Programming (ESOP'86)*, pages 119–132. Springer LNCS 213, 1986.

[11] A. Bockmayr, S. Krischer, and A. Werner. Narrowing strategies for arbitrary canonical systems. *Fundamenta Informaticae*, 24(1,2):125–155, 1995.

[12] G. Boudol. Computational semantics of term rewriting systems. In M. Nivat and J. C. Reynolds, editors, *Algebraic methods in semantics*, chapter 5, pages 169–236. Cambridge University Press, Cambridge, UK, 1985.

[13] P.H. Cheong and L. Fribourg. Implementation of narrowing: The Prolog-based approach. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic programming languages: constraints, functions, and objects*, pages 1–20. MIT Press, 1993.

[14] J. Darlington and Y. Guo. Narrowing and unification in functional programming - an evaluation mechanism for absolute set abstraction. In *Proc. Conference on Rewriting Techniques and Applications (RTA'89)*, pages 92–108. Springer LNCS 355, 1989.

[15] N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North Holland, Amsterdam, 1990.

[16] Nachum Dershowitz. Completion and its applications. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, chapter 2, pages 31–85. Academic Press, New York, 1989.

[17] R. Echahed. On completeness of narrowing strategies. *Theoretical Computer Science 72*, pages 133–146, 1990.

[18] R. Echahed. Uniform narrowing strategies. In *Proc. Third International Conference on Algebraic and Logic Programming (ALP'92)*, pages 259–275, Volterra, Italy, 1992. Springer LNCS 632.

[19] R. Echahed and J.C. Janodet. Admissible graph rewriting and narrowing. In *Proc. of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 325–340. MIT Press, June 1998.

[20] H. Fassbender and H. Vogler. A universal unification algorithm based on unification-driven leftmost outermost narrowing. *Acta Cybernetica*, 11(3):139–167, 1994.

[21] M. J. Fay. First-order unification in an equational theory. In *Proc. 4th Workshop on Automated Deduction*, pages 161–167, Austin (Texas), 1979. Academic Press.

[22] L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 172–184, Boston, 1985.

[23] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: a logic plus functional language. *The Journal of Computer and System Sciences*, 42:139–185, 1991.

[24] J. A. Goguen and J. Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pages 295–363. Prentice Hall, 1986.

[25] J.C. Gonzáles-Moreno, M.T. Hortalá-Gonzáles, F.J. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. European Symposium on Programming (ESOP'96)*, pages 156–172. Springer LNCS 1058, 1996.

[26] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 1978.

[27] M. Hamada and A. Middeldorp. Strong completeness of a lazy conditional narrowing calculus. In *Proc. 2nd Fuji International Workshop on Functional and Logic Programming*, pages 14–32. World Scientific, 1997.

[28] W. Hans, R. Loogen, and S. Winkler. On the interaction of lazy evaluation and backtracking. In *Proc. 4th International Symposium on Programming Language Implementation and Logic Programming (PLILP'92)*, pages 355–369. Springer LNCS 631, 1992.

[29] M. Hanus. Compiling logic programs with equality. In *Proc. 2nd Int. Workshop on Programming Language Implementation and Logic Programming (PLILP'90)*, pages 387–401. Springer LNCS 456, 1990.

[30] M. Hanus. Improving control of logic programs by using functional logic languages. In *Proc. 4th International Symposium on Programming Language Implementation and Logic Programming (PLILP'92)*, pages 1–23. Springer LNCS 631, 1992.

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

[32] M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation (LOPSTR'95)*, pages 252–266. Springer LNCS 1048, 1995.

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

[34] M. Hanus, S. Lucas, and A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8, 1998.

[35] M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. In *Proc. Seventh International Conference on Rewriting Techniques and Applications (RTA'96)*, pages 138–152. Springer LNCS 1103, 1996.

[36] M. Hanus and R. Sadre. An abstract machine for Curry and its concurrent implementation in Java. *Journal of Functional and Logic Programming*, (6), 1999.

[37] M. Hanus (ed.). Curry: An integrated functional logic language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1999.

[38] A. Herold. Narrowing techniques applied to idempotent unification. Technical Report SR-86-16, SEKI, 1986.

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

[40] G. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. *JCSS*, 25:239–266, 1982.

[41] G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*, pages 395–443. MIT Press, Cambridge, MA, 1991.

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

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

[44] J.A. Jiménez-Martín, J. Mariño-Carballo, and J.J. Moreno-Navarro. Efficient compilation of lazy narrowing into prolog. In *Proc. Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'92)*, pages 253–270. Springer Workshops in Computing Series, 1992.

[45] J. R. Kennaway. The specificity rule for lazy pattern-matching in ambiguous term rewrite systems. In *Proc. Third European Symp. on Programming (ESOP'90)*, pages 256–270. Springer LNCS 432, 1990.

[46] J. R. Kennaway, J. K. Klop, M. R. Sleep, and F. J. de Vries. The adequacy of term graph rewriting for simulating term rewriting. In M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors, *Term Graph Rewriting Theory and Practice*, pages 157–169. J. Wiley & Sons, Chichester, UK, 1993.

[47] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pages 1–112. Oxford University Press, 1992.

[48] J. W. Klop and A. Middeldorp. Sequentiality in orthogonal term rewriting systems. *Journal of Symbolic Computation*, pages 161–195, 1991.

[49] R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. 5th International Symposium on Programming Language Implementation and Logic Programming (PLILP'93)*, pages 184–200. Springer LNCS 714, 1993.

[50] L. Maranget. Optimal derivation in weak lambda-calculi and in orthogonal terms rewriting systems. In *18th Annual Symp. on Principles of Prog. Languages*, pages 255–269. ACM, 1991.

[51] A. Middeldorp and E. Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

[52] A. Middeldorp and S. Okui. A deterministic lazy narrowing calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

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

[54] J. J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy narrowing in a graph machine. In *Proc. Second International Conference on Algebraic and Logic Programming (ALP'90)*, pages 298–317. Springer LNCS 463, 1990.

[55] 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.

[56] J.J. Moreno-Navarro, H. Kuchen, J. Marino-Carballo, S. Winkler, and W. Hans. Efficient lazy narrowing using demandedness analysis. In *Proc. 5th International Symposium on Programming Language Implementation and Logic Programming (PLILP'93)*, pages 167–183. Springer LNCS 714, 1993.

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

[58] Robert Nieuwenhuis. On narrowing, refutation proofs and constraints. In Jieh Hsiang, editor, *Rewriting Techniques and Applications, 6th International Conferenc e, RTA-95*, LNCS 914, pages 56–70, Kaiserslautern, Germany, April 5–7, 1995. Springer-Verlag.

[59] W. Nutt, P. Réty, and G. Smolka. Basic narrowing revisited. *Journal of Symbolic Computation*, 7:295–317, 1989.

[60] M. J. O'Donnell. *Computing in Systems Described by Equations*. Springer LNCS 58, 1977.

[61] M. J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.

[62] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.

[63] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[64] G.D. Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, 1972.

[65] U. S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.

[66] R.C. Sekar and I.V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. *Information and Computation*, 104(1):78–109, 1993.

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

[68] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors. *Term Graph Rewriting Theory and Practice*. J. Wiley & Sons, Chichester, UK, 1993.

[69] J.J. Thiel. Stop losing sleep over incomplete data type specifications. In *11th ACM Symposium on Principles of Programming Languages*, pages 76–82, Salt Lake City, 1984.

[70] E. Ullán Hernández. A lazy narrowing abstract machine. Technical report DIA/95/3, Universidad Complutense, Madrid, 1995.

[71] P. Wadler. How to replace failure by a list of successes. In *Functional Programming and Computer Architecture*, pages 113–128. Springer LNCS 201, 1985.

[72] D.H.D. Warren. Higher-order extensions to prolog: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.

[73] J.-H. You. Unification modulo an equality theory for equational logic programming. *The Journal of Computer and System Sciences*, 42(1):54–75, 1991.