A Needed Narrowing Strategy

Sergio Antoy
Dept. of Computer Science
Portland State University
Portland, OR 97207
U.S.A.
antoy@cs.pdx.edu

Rachid Echahed
IMAG-LGI
CNRS
F-38041 Grenoble
France
echahed@imag.fr

Michael Hanus
MPI Informatik
Im Stadtwald
D-66123 Saarbrücken
Germany
michael@mpi-sb.mpg.de

Abstract

Narrowing is the operational principle of languages that integrate functional and logic programming. 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 independent unifiers, and is efficiently implemented by pattern matching.

1 Introduction

In recent years, most proposals with a sound and complete operational semantics for the integration of functional and logic programming languages [5, 10] were based on narrowing, e.g., [6, 15, 17, 19, 37, 44]. Narrowing, originally introduced in automated theorem proving [46], solves equations by computing unifiers with respect to an equational theory [14]. Informally, narrowing unifies a term with the left-hand side of a rewrite rule and fires the rule on the instantiated term.

Example 1 Consider the following rewrite rules defining the operations “less than or equal to” and addition for natural numbers, which are represented by terms built with 0 and s:

\[
\begin{align*}
0 \leq X & \rightarrow \text{true} & \text{R}_1 \\
 s(X) \leq 0 & \rightarrow \text{false} & \text{R}_2 \\
 s(X) \leq s(Y) & \rightarrow X \leq Y & \text{R}_3 \\
 0 + X & \rightarrow X & \text{R}_4 \\
 s(X) + Y & \rightarrow s(X + Y) & \text{R}_5
\end{align*}
\]

The rules of “≤” will be used in following examples. To narrow the equation \( Z + s(0) \approx s(s(0)) \), rule \( \text{R}_3 \) is applied by instantiating \( Z \) to \( s(X) \). To narrow the resulting equation \( s(X + s(0)) \approx s(s(0)) \), \( \text{R}_4 \) is applied by instantiating \( X \) to 0. The resulting equation, \( s(s(0)) \approx s(s(0)) \), is trivially true. Thus, \( \{ Z \rightarrow s(0) \} \) is the equation’s solution.

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. 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 [25], innermost [15], outermost [12], outer [49], lazy [9, 36, 44], or narrowing with redundancy tests [31]. 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.)

Our contribution is a strategy that, for inductively sequential systems [1], 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 independence 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 \( t \) [24, 30, 39]. 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.

Restricting our discussion to this subclass is not a limitation for the use of narrowing in programming lan-
gures. Computing a needed redex in a term is an unsolvable problem. Strongly sequential systems are, in practice, the largest class for which the problem becomes solvable. Inductively sequential systems are a large constructor-based subclass of the strongly sequential systems.

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. We compare related work in Section 6. Our conclusion is in Section 7. Due to lack of space we omit the proofs of the theorems, but the interested reader will find them in [3].

2 Preliminaries

We recall some key notions and notations about rewriting. See [11, 29] for tutorials.

Terms are constructed w.r.t. a given many-sorted signature \( \Sigma \). We write \( \text{Var}(t) \) for the set of variables occurring in a term \( t \). Equational logic programs are generally constructor-based, i.e., 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 [40] and the functional logic languages ALF [19], BABEL [37], K-LEAF [16], LPG [6], SLOG [15]). Hence, we assume that \( \mathcal{R} \) is a constructor-based term rewriting system consisting of rewrite rules of the form \( l \to r \), where \( l \) is an innermost term, i.e., the root of \( l \) is an operation and the arguments of \( l \) do not contain any operation symbol.

Substitutions and unifiers are defined as usual [11], where we write \( \text{mgu}(s, t) \) for the most general unifier of \( s \) and \( t \). We write \( \sigma \leq \sigma’[V] \) iff there is a substitution \( \tau \) with \( \sigma’(x) = \tau(\sigma(x)) \) for all variables \( x \in V \). Two substitutions \( \sigma \) and \( \sigma’ \) are independent on a set of variables \( V \) iff there exists some \( x \in V \) such that \( \sigma(x) \neq \sigma’(x) \) are not unifiable.

An occurrence or position \( p \) is a path identifying a subterm in a term \( t \). \( t|_p \) denotes the subterm of \( t \) at position \( p \), and \( t[s]_p \) denotes the result of replacing \( t|_p \) with \( s \) in \( t \).

A term rewriting system \( \mathcal{R} \) is orthogonal if for each rule \( l \to r \in \mathcal{R} \) the left-hand side \( l \) does not contain multiple occurrences of one variable (left-linearity) and for each non-variable subterm \( t|_p \) of \( l \) there exists no rule \( l’ \to r’ \in \mathcal{R} \) such that \( t|_p \) and \( l’ \) unify (non-overlapping).

A rewrite step \( t \overset{\rho}{\to} t’ \) is the application of the rule \( l \to r \) to the redex \( t|_p \), i.e., \( s = t[\sigma(r)]_p \) for some substitution \( \sigma \) with \( t|_p = \sigma(l) \). A term is in normal form if it cannot be rewritten. Functional logic programs compute with partial information, i.e., a functional expression may contain logical 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 [16, 37]. This is done by narrowing.

Definition 1 A term \( t \) is narrowable to a term \( s \) if there exist a non-variable position \( p \) in \( t \) (i.e., \( t|_p \) is not a variable), a variant \( l \to r \) of a rewrite rule in \( \mathcal{R} \) with \( \text{Var}(t) \cap \text{Var}(l \to r) = \emptyset \) and a unifier \( \sigma \) of \( t|_p \) and \( l \) such that \( s = \sigma(t|_p) \). In this case we write \( t \overset{p}{\sim} t \to r \sigma \) or \( t \overset{p}{\sim} _t \to r \sigma \). If \( \sigma \) is a most general unifier of \( t|_p \) and \( l \), the narrowing step is called most general. We write \( t_0 \overset{p_1}{\sim} \ldots \overset{p_n}{\sim} t_n \) if there is a narrowing sequence \( t_0 \overset{p_1}{\sim} t_1 \overset{p_2}{\sim} t_2 \overset{\sigma_2}{\sim} \ldots \overset{p_{n-1}}{\sim} \sigma_{n-1} t_n \) with \( \sigma = \sigma_0 \circ \cdots \circ \sigma_{n-1} \).

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

Example 2 Referring to Example 1,

\[
A + B \overset{\lambda \in A, R_5, \{ A \to s(0) \}, B_0 = s(1)}{\sim} s(0 + 0)
\]

and

\[
A + B \overset{\lambda \in A, R_5, \{ A \to s(X) \}}{\sim} s(X + B)
\]

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

Padawitz [42] too distinguishes between narrowing and most general narrowing, but in most papers narrowing is intended as most general narrowing (e.g., [25]). Most general narrowing has the advantage that most general unifiers are uniquely computable, whereas there exist many independent unifiers. Dropping the requirement that unifiers be most general is crucial to the definition of 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. Since we do not require terminating term rewriting systems, normal forms may not exist. 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 [16] and BABEL [37]. Thus, a substitution \( \sigma \) is a solution for an equation \( t \approx t’ \) iff \( \sigma(t) \approx \sigma(t’) \) are reducible to a same ground constructor term. 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 \( \approx \) is defined by the following rules, where \( \land \) 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.

\[
c \approx c \quad \to \quad \text{true}
\]

\[
c(X_1, \ldots, X_n) \approx c(Y_1, \ldots, Y_n) \quad \to \quad \bigwedge_{i=1}^n (X_i \approx Y_i)
\]

\[
\text{true} \land X \quad \to \quad X
\]
These are the equality rules of a signature. It is easy to see that the orthogonality status of a rewrite system is not changed by these rules. The same holds true for the inductive sequentality, 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 [16] and BABEL [37]. 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.

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 [24] formalizes this idea.

**Definition 2** Let \( A = t \rightarrow_{l \rightarrow r} t' \) be a rewrite step of some term \( t \) into \( t' \) at position \( v \) with rule \( l \rightarrow r \). The set of **descendants** (or residuals) of a position \( v \) by \( A \), denoted \( v \setminus A \), is

\[
v \setminus A = \begin{cases} 
\{ v \} & \text{if } u = v, \\
\{ u'q \text{ such that } r_{l'v} = x \} & \text{if } u \not= v, \\
\text{if } v = upq \text{ and } l'p = x, \\
\text{where } x \text{ is a variable. }
\end{cases}
\]

The set of **descendants** of a position \( v \) by a rewrite derivation \( B \) is defined by induction as follows

\[
v \setminus B = \begin{cases} 
\{ v \} & \text{if } B = \emptyset, \\
\bigcup_{w \in v \setminus B} w \setminus B^n & \text{if } B = B'B^e.
\end{cases}
\]

A position \( u \) of a term \( t \) is called **needed** if in every rewrite derivation 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 [30]. Let \( t \rightarrow t' \) 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 \( t \rightarrow t' \). Then, every subterm of \( t' \) with an underlined root is a **descendant** of \( s \).

**Example 3** Consider the operation that doubles its argument by means of an addition. The rules of addition are in Example 1.

\[
double(X) \rightarrow X + X \quad R_d
\]

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

\[
double(0 + 0) \rightarrow_{\lambda, R_d} (0 + 0) + (0 + 0)
\]

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 [25] has introduced basic narrowing, where narrowing is not applied at positions introduced by substitutions. Fribourg [15] has proposed innermost narrowing, where narrowing is applied only at an innermost position, and Hölldöbler [22] 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 [12]. Lazy narrowing [9, 36, 44], 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, and 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 \( t \sim_{p, l \rightarrow r, \sigma} t' \) is needed iff \( p \) is a position of \( t \). \( \sigma \) 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 4** Referring to Example 1, consider the term \( t = X \leq Y + Z \). According to the naive approach, the following narrowing step of \( t \) at position 2

\[
X \leq Y + Z \sim_{2, R_4 \atop \{Y \rightarrow 0\}} X \leq Z
\]

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 note. However, the same step becomes unnecessary if \( 0 \) is substituted for \( X \), as shown by the following derivation, which computes a more general solution of the inequality without ever narrowing any descendant of \( t \) at 2.

\[
X \leq Y + Z \sim_{\lambda, R_1 \atop \{X \rightarrow 0\}} true
\]

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 3 A narrowing step \( t \sim_{R, \sigma} t' \) 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 [12, 42], 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 4 shows, are essential to capture the need of a narrowing step.

Definition 4 A narrowing strategy is a function from terms into sets of triples. If \( S \) is a narrowing strategy, \( t \) is a term, and \( (p, l \rightarrow r, \sigma) \in S(t) \), then \( p \) is a position of \( t \), \( l \rightarrow r \) is a rewrite rule, and \( \sigma \) a substitution such that \( t \sim_{p, l \rightarrow r, \sigma} \sigma(t[r]_p) \) 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 [1], which is used to implement needed rewriting. This note generalizes that result to narrowing.

The symbols branch, rule, and exempt, used in the next definition, are uninterpreted functions used to classify the nodes of the tree. A pattern is an innermost term contained in each node.

Definition 5 \( \mathcal{T} \) is a partial definitional tree, or pdt, with pattern \( \pi \) w.r.t. a constructor-based rewrite system \( \mathcal{R} \) iff one of the following cases holds:

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

\[
\mathcal{T} = \text{rule}(\pi, l \rightarrow r), \text{ where } \pi \text{ is a pattern and } l \rightarrow r \text{ is a rewrite rule in } \mathcal{R} \text{ such that } l = \pi.
\]

\[
\mathcal{T} = \text{exempt}(\pi), \text{ where } \pi \text{ is a pattern and } l \not\subseteq \pi \text{ for every rule } l \rightarrow r \text{ in } \mathcal{R}.
\]

\( \mathcal{T} \) is a definitional tree of an operation \( f \) iff \( \mathcal{T} \) is a pdt with \( f(X_1, \ldots, X_n) \) as the pattern argument, where \( n \) is the arity of \( f \) and \( X_1, \ldots, X_n \) are new variables.

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 the rules contained in \( \mathcal{T} \) are all and only the rules defining \( f \) in \( \mathcal{R} \). We call inductively sequential a rewrite system \( \mathcal{R} \) iff any operation of \( \mathcal{R} \) is inductively sequential.

Example 5 We show a pictorial representations of definitional trees of the operations defined in Example 1. A branch node of the picture shows the pattern of a corresponding node of the definitional tree. A leaf node of the picture shows the right sides of a rule contained in a rule node of the tree. The occurrence argument of a branch node is shown by embedding the corresponding subterm in the pattern argument.

\[
\begin{align*}
Y_1 + Y_2 & \quad 0 + Y_2 \\
\downarrow & \quad s(Y_3) + Y_2 \\
Y_2 & \quad \text{true} \\
\downarrow & \quad s(X_3) \leq 0 \\
X_1 \leq X_2 & \quad \text{false} \\
\downarrow & \quad X_3 \leq X_4
\end{align*}
\]

Inductively sequential systems are constructor-based and strongly sequential [1]. We conjecture that these two classes are the same. 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 [28]. We now give an informal account of our strategy.
The computation of a definitional tree is a finite set process.

The pattern of a definitional tree is a finite set process.

Definition 6. The function \( \lambda \) takes two arguments

\( \lambda : T \times T \rightarrow \mathbb{N} \)

and returns a number in the range \( 0 \) through \( \lambda \).

\( \lambda (\alpha, \beta) = \begin{cases} 
0 & \text{if } \alpha = \beta \\
1 & \text{if } \alpha \neq \beta \end{cases} \)

The function \( \lambda \) is an operation-flag set.

A function \( \lambda \) is a partial function if it is not defined on all pairs of arguments.

Example 6. We define a pattern \( \lambda \) for the infinite tree \( T \).

\( \lambda (\alpha, \beta) = \begin{cases} 
0 & \text{if } \alpha = \beta \\
1 & \text{if } \alpha \neq \beta \end{cases} \)

The function \( \lambda \) is a partial function if it is not defined on all pairs of arguments.

We define a pattern \( \lambda \) for the infinite tree \( T \).

\( \lambda (\alpha, \beta) = \begin{cases} 
0 & \text{if } \alpha = \beta \\
1 & \text{if } \alpha \neq \beta \end{cases} \)

The function \( \lambda \) is a partial function if it is not defined on all pairs of arguments.

We define a pattern \( \lambda \) for the infinite tree \( T \).

\( \lambda (\alpha, \beta) = \begin{cases} 
0 & \text{if } \alpha = \beta \\
1 & \text{if } \alpha \neq \beta \end{cases} \)

The function \( \lambda \) is a partial function if it is not defined on all pairs of arguments.

We define a pattern \( \lambda \) for the infinite tree \( T \).

\( \lambda (\alpha, \beta) = \begin{cases} 
0 & \text{if } \alpha = \beta \\
1 & \text{if } \alpha \neq \beta \end{cases} \)

The function \( \lambda \) is a partial function if it is not defined on all pairs of arguments.

We define a pattern \( \lambda \) for the infinite tree \( T \).

\( \lambda (\alpha, \beta) = \begin{cases} 
0 & \text{if } \alpha = \beta \\
1 & \text{if } \alpha \neq \beta \end{cases} \)

The function \( \lambda \) is a partial function if it is not defined on all pairs of arguments.

We define a pattern \( \lambda \) for the infinite tree \( T \).

\( \lambda (\alpha, \beta) = \begin{cases} 
0 & \text{if } \alpha = \beta \\
1 & \text{if } \alpha \neq \beta \end{cases} \)

The function \( \lambda \) is a partial function if it is not defined on all pairs of arguments.

We define a pattern \( \lambda \) for the infinite tree \( T \).

\( \lambda (\alpha, \beta) = \begin{cases} 
0 & \text{if } \alpha = \beta \\
1 & \text{if } \alpha \neq \beta \end{cases} \)

The function \( \lambda \) is a partial function if it is not defined on all pairs of arguments.
\[
\lambda(t, \mathcal{T}) \equiv \begin{cases} 
(\lambda, R, mgu(t, \pi)) & \text{if } \mathcal{T} = \text{rule}(\pi, R); \\
(\lambda, ?, mgu(t, \pi)) & \text{if } \mathcal{T} = \text{exempt}(\pi); \\
(p, R, \sigma) & \text{if } \mathcal{T} = \text{branch}((\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\
t \text{ and pattern}(\mathcal{T}_i) \text{ unify, for some } i, \text{ and} \\
(p, R, \sigma) \in \lambda(t, \mathcal{T}_i); \\
(o \cdot p, R, \sigma \circ \tau) & \text{if } \mathcal{T} = \text{branch}((\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\
t \text{ and pattern}(\mathcal{T}_i) \text{ do not unify, for any } i, \\
\tau = mgu(t, \pi), \\
\mathcal{T}' \text{ is a definitional tree of the root of } \tau(t|_{\mathcal{T}}), \text{ and} \\
(p, R, \sigma) \in \lambda(\tau(t|_{\mathcal{T}}), \mathcal{T}'). 
\end{cases}
\]

Figure 1: Definition of \(\lambda\)

Essential, since to narrow a term “all the way” a strategy should compute a narrowing step, when one exists. Indeed, in incomplete rewrite systems, \(\lambda\) may fail to compute any narrowing step even when some step could be computed.

Example 7 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 \(1\). 1, but \(\lambda\) returns the set \(\{(1, ?, \{\})\}\).

The inability of \(\lambda\) 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 [7] by another narrowing step. If \((p, ?, \sigma) \in \lambda(t, \mathcal{T})\), no equation having \(\sigma(t)\) as one side can be solved. Any amount of work applied toward finding a solution would be wasted. 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 one side may even have an infinite search space. However, any amount of work applied toward finding a solution would be wasted.

Example 8 Consider the following term rewriting system for subtraction:

\[
\begin{align*}
X - 0 & \rightarrow X & \text{R}_1 \\
s(X) - s(Y) & \rightarrow X - Y & \text{R}_2
\end{align*}
\]

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 \(0 - s(X) \approx Y - Z\) while there would be 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 [1]. 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 8 has an exempt node for the pattern \(0 - s(X)\). However, Definition 5 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 can be avoided if all branch nodes in a definitional tree are useful, i.e., there is at least one rule node in each branch subpath.

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 \(\lambda\) 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 [1, 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 $\lambda$. 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’.

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 $t \approx t'$ is true, then $\sigma$ is an outermost-needed narrowing derivation, then $\sigma$ is a solution for $t \approx t'$.

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 irreducible solutions [42], and lazy narrowing is complete only for constructor substitutions [16, 37]. The following theorem shows the completeness of our strategy, $\lambda$, and consequently of outermost-needed narrowing.

**Theorem 3** (Completeness of outermost-needed narrowing) Let $\mathcal{R}$ be an inductively sequential rewrite system extended by the equality rules. Let $\sigma$ be a constructor substitution that is a solution of an equation $t \approx t'$ and $V$ be a finite set of variables containing $\text{Var}(t) \cup \text{Var}(t')$. Then there exists a derivation $t \approx t', t' \approx t$, true computed by $\lambda$ such that $\sigma' \leq \sigma[V]$.

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, and 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 length. The next theorem claims that no redundant derivation is computed by $\lambda$.

**Theorem 4** (Independence of solutions) Let $\mathcal{R}$ be an inductively sequential rewrite system extended by the equality rules. Let $e \sim_{\sigma} e'$ true and $e \sim_{\sigma'} e''$ true be two distinct derivations computed by $\lambda$. Then, $\sigma$ and $\sigma'$ are independent on $V$.

We now discuss the cost and length of a derivation computed by our strategy.

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, e.g., see Example 3. However, from a practical standpoint, if terms are appropriately represented, the cost of narrowing $t$ (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.

**Definition 7** Let $t \sim_{p; I \rightarrow \Psi, \sigma} t'$, 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 $t \sim_{p; I \rightarrow \Psi, \sigma} t'$, iff $t' = \circ_{i \in I} \sigma^i((t(r^1_{\Psi})[^2_{\Psi}] \ldots [^n_{\Psi}]),$ where $\circ_{i \in I} \sigma^i$ denotes the composition $\sigma^0 \circ \sigma^1 \circ \sigma^2 \circ \sigma^3 \circ \sigma^4$ (the order is irrelevant).

When we want to emphasize the difference between a step as defined in Definition 1 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.

To claim that our strategy is optimal, we assign a “cost” to both a step and a derivation. By convention, 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. We will justify our choice after giving our definition of cost.
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 8** Let \( \alpha = t_0 \sim_{\sigma} \cdots \sim_{\sigma'} t_n \) be a narrowing (multi)derivation. 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 \( o \in I_{n+1} \sigma_{n+1}^k (t_n |_{\sigma_{n+1}^k}) \) and \( o \in I_{n+1} \sigma_{n+1}^k (t_n |_{\sigma_{n+1}^k}) \) are descendants of \( o \in I_{n+1} \sigma_{n+1}^k (t_m |_{\sigma_{n+1}^k}) \).

We call a family any maximal subset of equivalent indices. The cost of the \( n \)-th step of \( \alpha \) is the number of families in \( I_n \), i.e., \([I_n]/\sim_n\). The cost of \( \alpha \), denoted \( cost(\alpha) \), is the total cost of its steps.

We say that a family is complete iff it cannot be enlarged, and we say that a step is complete iff all its families are complete, more precisely \( I_n \) is complete iff \( i \in I_n \) then for any position \( q \) of \( o \in I_n \sigma_{n+1}^k (t_n |_{\sigma_{n+1}^k}) \) such that \( p_i \) and \( q \) have a common ancestor in some term of \( \alpha \), there exists some \( j \) in \( I_n \) such that \( q = p_j \).

We say that a derivation is complete iff all its steps are complete.

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 \( t \sim_{p} \sigma \), \( t \sigma \sigma \), and \( q \sigma \) and \( p \sigma \) 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 the substituting term (the instantiation of the right side of a rule.) since the reduces of blood related subterms are all equal, too. This implies that all the members of a family could “be shared” in the representation of \( t \). When this is being done (as in efficient implementations of narrowing [19]), a multistep entailing a whole family does not differ in practice, from an elementary step.

**Theorem 5** If \( \alpha = t \sim_{\sigma} u \) is a complete outermost needed narrowing multiderivation of a term \( t \sigma \) into a constructor term \( u \), then \( \alpha \) has minimum cost. I.e., for any multiderivation \( \beta = t \sim_{\sigma} u \), \( cost(\alpha) \leq cost(\beta) \).

Elementary steps are easier to understand and to implement than multisteps. To achieve optimality, we need multisteps only as far as blood related terms are concerned. Full sharing of blood related subterms implies that no family ever contains more than a single member, in practice, and thus any elementary step becomes trivially complete. In turn, this equates derivations of minimum cost with those of minimum length. Techniques for rewriting “terms” with shared subterms go under the name of term graph rewriting [47] and adapting them to narrowing, for the systems we are considering, poses no major problem [4].

### 6 Related work

There are three research topics related to our work: (1) the concept of need as the foundation of laziness, (2) strategies for using narrowing in programming, and (3) implementations of narrowing in Prolog.

#### 6.1 Narrowing and need

Seminal studies on the concept of need in rewriting appear in [24, 39]. Subsequent variations and extensions, e.g., [7, 21, 27, 30, 33, 40, 41, 45, 48], do not address narrowing, but it is still relevant. We have introduced a concept of need for narrowing that extends a similar concept for rewriting. We have shown that the concept of need for narrowing is inherently more complicated than that for rewriting. In orthogonal systems, a reduction step has one degree of freedom: the selection of the position, but a narrowing step has two. Both the position and the unifier. We have discussed only inductively sequential systems. Further research will extend this class to strongly sequential and/or weakly orthogonal systems. The extension to weakly orthogonal systems would weaken our strong optimality result, but include additional non-determinism. Sekar and Ramakrishnan [45] propose necessary sets as a generalization of the notion of need for weakly orthogonal systems. Antoy [1] suggests rewriting necessary sets of redexes using parallel definition trees and a function analogous to \( \lambda \). This approach can be extended to narrowing without major problems.

#### 6.2 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., [9, 12, 13, 14, 15, 16, 18, 20, 22, 31, 35, 36, 37, 38, 44, 49] have been proposed. The notion of completeness has evolved accordingly. Plotkin’s classic formulation [43] has been relaxed to completeness w.r.t. ground solutions (e.g., [15]) or completeness w.r.t. strict equality and domain-based interpretations, as in [16, 37]. 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.

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

**Example 9** The symbols $a$, $b$, and $c$ are constructors, whereas $f$ and $g$ are defined operations.

$$
\begin{align*}
  f(a) & \rightarrow a & R_1 \\
  f(b(X)) & \rightarrow b(f(X)) & R_2 \\
  f(c(X)) & \rightarrow a & R_3 \\
  g(a, X) & \rightarrow b(a) & R_4 \\
  g(b(X), a) & \rightarrow a & R_5 \\
  g(b(X), b(Y)) & \rightarrow c(a) & R_6 \\
  g(c(X), c(Y)) & \rightarrow b(a) & R_7 \\
  g(c(X), Y) & \rightarrow b(a) & R_8
\end{align*}
$$

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

$$
\begin{align*}
  g(X, f(X)) & \simeq c(a) \sim_{1, R_4, \{X \rightarrow e\}} b(a) \simeq c(a) \\
  g(X, f(X)) & \simeq c(a) \sim_{1, R_5, \{X \rightarrow c(X_0)\}} b(a) \simeq c(a) \\
  g(X, f(X)) & \simeq c(a) \sim_{1, 2, R_7, \{X \rightarrow d(X_1)\}} \\
  g(b(X_1), b(f(X_1))) & \simeq c(a) \sim_{1, 1} \text{true}
\end{align*}
$$

**Basic narrowing** [25] avoids positions introduced by the instantiations of previous steps. Its completeness, and that of its variations, e.g., [20, 22, 31, 35, 38], is known for convergent rewrite systems (see [35] for a systematic study.) This strategy may perform useless steps and computes an infinite search space for our benchmark example.

**Innermost narrowing** [15] narrows only innermost terms. It is ground complete only for strongly 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** [12, 13] narrows outermost operation-rooted terms. This strategy is complete only for a restrictive class of rewrite systems. It computes no solution for our benchmark example.

**Outer narrowing** [49] selects an inner position only when a step at an outer position is impossible. This strategy is complete for constructor-based systems. Outer narrowing behaves as needed narrowing on the benchmark example, however the strategy is not characterized as computing needed steps. Furthermore, [49] describes the enumeration of derivations for E-matching, but not the computation of derivations for general E-unification.

**Lazy narrowing** [9, 16, 18, 37, 36, 44], similar to outer, 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: [18] requires an ad-hoc implementation of backtracking, with the potential of evaluating some term several times; [16] requires flattening of functional nesting and a specialized WAM-like machine in which terms are dynamically reordered; [37] requires a transformation of the rewrite system which, for our benchmark example, increases the number of operations and lengthen the derivations.

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.

### 6.3 Narrowing in Prolog

Implementations of narrowing in Prolog [2, 8, 26, 32] are proposed as a prototypical and portable integration of functional and logic languages. For example, [8, 26] have been proposed as an alternative to the specialized machines required for $K$-LEAF [16] and $BABEL$ [37] respectively. The most recent proposals [2, 32] 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 [2] computes $\lambda$ directly by pattern matching. The patterns involved in the computation of $\lambda$ are a superset of those contained in a definitional tree. This is suggested by claim I of Theorem 1 that shows a “strong” need for the positions computed using $\lambda$—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.

### 7 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 a potentially infinite number of substitutions. Remarkably, there is an efficiently algorithm for this computation in inductively sequential systems.
We have contained our discussion to narrowing operation-rooted terms. This limitation shortens our discussion and suffices for solving equations. Extending our results also 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 have shown how our strategy is easily implemented by pattern matching, and we have reported, in the previous section, its good performance in Prolog with respect to other similar attempts. We have also shown that our strategy computes only independent 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 prove both completeness and optimality for inductively sequential systems. We believe that it is possible to extend our result to strongly sequential systems and, in a weaker form, to weakly orthogonal systems.

Acknowledgement

Aart Middendorp suggested us how to prove [34] our conjecture that the classes of inductively sequential systems and constructor-based strongly sequential systems are the same.

We would like to acknowledge the support of The Oregon Center for Advanced Technology Education (OCATE) for parts of the collaborative efforts that lead to the writing of this paper.

The research of Michael Hanus was supported in part by the German Ministry for Research and Technology (BMFT) under grant ITS 9103 and by the ESPRIT Basic Research Working Group 6028 (Construction of Computational Logics).

References


