In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, Portland, 1994

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:

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

The rules of "$\leq$" 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.

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 languages. 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* Σ. We write $\mathcal{V}ar(t)$ 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 $mgu(s,t)$ for the *most general unifier* of s and t. We write $\sigma \leq \sigma'[V]$ iff there is a substitution τ with $\sigma'(x) = \tau(\sigma(x))$ for all variables $x \in V$. Two substitutions σ and σ' are *independent* on a set of variables V iff there exists some $x \in V$ such that $\sigma(x)$ and $\sigma'(x)$ are not unifiable.

An *occurrence* or *position* p is a path identifying a subterm in a term. $t|_p$ 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 $l|_p$ of l there exists no rule $l' \to r' \in \mathcal{R}$ such that $l|_p$ and l' unify (*non-overlapping*).

A *rewrite step* $t \to_{p,l\to r} s$ is the application of the rule $l \to r$ to the *redex* $t|_p$, i.e., $s = t[\sigma(r)]_p$ for some substitution σ 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 $\mathcal{V}ar(t) \cap \mathcal{V}ar(l \to r) = \emptyset$ and a unifier σ of $t|_p$ and l such that $s = \sigma(t[r]_p)$. In this case we write $t \leadsto_{p,l\to r,\sigma} s$. If σ is a most general unifier of $t|_p$ and l, the narrowing step is called *most general*. We write $t_0 \overset{*}{\leadsto}_\sigma t_n$ if there is a narrowing sequence $t_0 \leadsto_{p_1,R_1,\sigma_1} t_1 \leadsto_{p_2,R_2,\sigma_2} \cdots \leadsto_{p_n,R_n,\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_2 \circ \sigma_1$.

Since the instantiation of the variables in the rule $l \to r$ 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 2 Referring to Example 1,

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

and

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

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

Padawitz [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 σ is a *solution* for an equation $t \approx t'$ iff $\sigma(t)$ and $\sigma(t')$ 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 $\wedge$ is assumed to be a right-associative infix symbol, and c is a constructor of arity 0 in the first rule and arity $n > 0$ is the second rule.

$$\begin{array}{rcl} c \approx c & \to & \mathit{true} \\ c(X_1,\ldots,X_n) \approx c(Y_1,\ldots,Y_n) & \to & \wedge_{i=1}^n (X_i \approx Y_i) \\ \mathit{true} \wedge X & \to & X \end{array}$$

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 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* [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 $A = t \to_{u,l\to r} t'$ be a rewrite 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

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

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

$$v \setminus B = \begin{cases} \{v\} & \text{if } B = \emptyset, \\ \bigcup_{w \in v\setminus B'} w \setminus B'' & \text{if } B = B'B''. \end{cases}$$

A position u of a term t is called *needed* iff 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 $t \xrightarrow{*} t'$ 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 $t \xrightarrow{*} t'$. 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.

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

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

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

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 [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 Hölldobler [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 $t \rightsquigarrow_{p,l\to r,\sigma} t'$ 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 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

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

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 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 inequation without ever narrowing any descendant of t at 2.

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

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 $t \rightsquigarrow_{p,R,\sigma} t'$ 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 [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 $\mathcal{S}$ 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 [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.

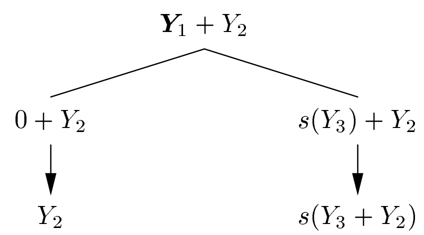
Definition 5 $\mathcal{T}$ is a *partial definitional tree*, or *pdt*, with pattern π w.r.t. a constructor-based rewrite system $\mathcal{R}$ iff one of the following cases holds:

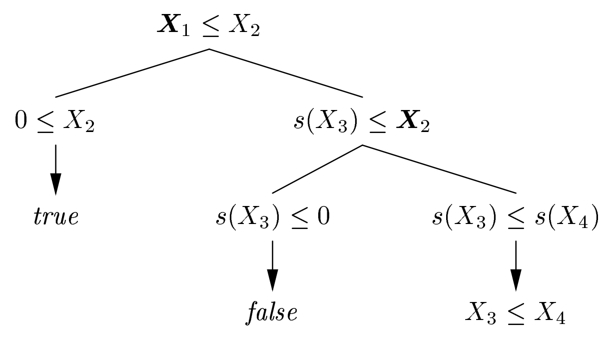
- $\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, where π is a pattern, o is the occurrence of a variable of π, the sort of $\pi|_o$ has constructors $c_1, \ldots, c_k$, for some $k > 0$, and for all i in $\{1, \ldots, k\}$, $\mathcal{T}_i$ is a *pdt* with pattern $\pi[c_i(X_1, \ldots, X_n)]_o$, where n is the arity of c_i and $X_1, \ldots, X_n$ are new variables.
- $\mathcal{T} = rule(\pi, l \to r)$, where π is a pattern and $l \to r$ is a rewrite rule in $\mathcal{R}$ such that $l = \pi$.
- $\mathcal{T} = exempt(\pi)$, where π is a pattern and $l \not\leq \pi$ for every rule $l \to r$ 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 emboldening the corresponding subterm in the pattern argument.





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 patterns of a definitional tree are a finite set partially ordered by the subsumption preordering and complete in the sense of [23]. 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 is contained. If $\mathcal{T}$ is a *rule pdt*, then we narrow $\tau(t)$ at the root with the rule contained in $\mathcal{T}$. 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 two arguments, 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 definition, $pattern(\mathcal{T})$ denotes the pattern argument of $\mathcal{T}$.

Definition 6 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 strong arithmetical induction on the number of occurrences of operation symbols in t and by structural induction on $\mathcal{T}$ in Figure 1. The function λ is well-defined in the third case since, 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 innermost term. Since t is operation-rooted and $o \neq \Lambda$, $\tau(t|_o)$ has fewer occurrences of 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 too.

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 constructor-rooted or 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 6 We trace the computation of λ for the initial step of a derivation of $X \leq Y + Z$, which was discussed in Example 4.

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

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 [24], is extended to narrowing simply by replacing $\to_{u,l\to r}$ with $\leadsto_{u,l\to r,\sigma}$ in Definition 2.

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 $R = l \to r$, then $t \leadsto_{p,R,\sigma} \sigma(t[r]_p)$ is an outermost-needed narrowing step.*
3. *If $R = ?$, then $\eta(t)$ cannot be narrowed to a constructor-rooted term.*

We say that a narrowing derivation is *computed by* λ iff for each step $t \leadsto_{p,R,\sigma} t'$ 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

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

Figure 1: Definition of λ

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

$$f(0) \to 0$$

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 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* [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 a 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:

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

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 $0 - s(X) \approx Y - Z$ 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* sub*pdt*.

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 [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 $\wedge$. 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'.*

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 $t \approx t'$.*

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, λ, 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 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 $\sigma' \leq \sigma[V]$.*

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

Theorem 4 (Independence 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 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 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.

Definition 7 Let $t \rightsquigarrow_{p^i, l^i \to r^i, \sigma^i} t^i$, 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 $t \rightsquigarrow_{\langle p^i, l^i \to r^i, \sigma^i \rangle_{i \in I}} t'$, 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 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 $\alpha = t_0 \rightsquigarrow_{\langle p_1^i, R_1^i, \sigma_1^i \rangle_{i \in I_1}} t_1 \rightsquigarrow_{\langle p_2^i, R_2^i, \sigma_2^i \rangle_{i \in I_2}} \cdots$ 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 $\circ_{k \in I_{n+1}} \sigma_{n+1}^k (t_n|_{p_{n+1}^i})$ and $\circ_{k \in I_{n+1}} \sigma_{n+1}^k (t_n|_{p_{n+1}^j})$ are descendants of $\circ_{k \in I_{m+1}} \sigma_{m+1}^k (t_m|_q)$.

We call a *family* any maximal subset of equivalent indices. 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 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 if i is in I_n, then for any position q of $\circ_{k \in I_n} \sigma_{n+1}^k (t_{n-1})$ 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.

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 q and p 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 side of a rule,) since the reducts 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 $\alpha = t \overset{*}{\rightsquigarrow}_\sigma u$ is a complete outermost-needed narrowing multiderivation of a term t into a constructor term u, then α has minimum cost. I.e., for any multiderivation $\beta = t \overset{*}{\rightsquigarrow}_\sigma u$, $cost(\alpha) \leq cost(\beta)$.*

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 limit the discussion to rewriting. 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* definitional trees and a function analogous to λ. 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.

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

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.

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

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 λ directly by pattern matching. The patterns involved in the computation of λ are a superset of those contained in 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.

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 Middeldorp 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

[1] S. Antoy. Definitional trees. In *ALP'92*, pages 143–157. Springer LNCS 632, 1992.

[2] S. Antoy. Lazy rewriting in logic programming. Technical Report 90-17, Rev. 2, Portland State University, Portland, OR, 1992. (Submitted for publication).

[3] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. Technical report, MPI-I-93-243, Max-Planck-Institut für Informatik, Saarbrücken, 1993.

[4] H. Barendregt, M. van Eekelen, J. Glauert, R. Kenneway, and M. Sleep. Term graph rewriting. In *PARLE'87*, pages 141–158. Springer LNCS 259, 1987.

[5] M. Bellia and G. Levi. The relation between logic and functional languages: a survey. *Journal of Logic Programming*, 3(3):217–236, 1986.

[6] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *ESOP-86*, pages 119–132. Springer LNCS 213, 1986.

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

[8] P. H. Cheong. Compiling lazy narrowing into Prolog. *New Generation Computing*, 1992. (to appear).

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

[10] D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations, and Equations.* Prentice Hall, 1986.

[11] 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.

[12] R. Echahed. On completeness of narrowing strategies. In *Proc. CAAP'88*, pages 89–101. Springer LNCS 299, 1988.

[13] R. Echahed. Uniform narrowing strategies. In *Proceedings of the Third International Conference on Algebraic and Logic Programming*, pages 259–275, Volterra, Italy, September 1992.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] M. Hanus. Compiling logic programs with equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pages 387–401. Springer LNCS 456, 1990.

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

[21] C. M. Hoffmann and M. J. O'Donnell. Implementation of an interpreter for abstract equations. In *11th ACM Symposium on the Principle of Programming Languages*, Salt Lake City, 1984.

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

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

[24] 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.* MIT Press, Cambridge, MA, 1991. Previous version: Call by need computations in non-ambiguous linear term rewriting systems, Technical Report 359, INRIA, Le Chesnay, France, 1979.

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

[26] J. A. Jiménez-Martín, J. Mariño-Carballo, and J. J. Moreno-Navarro. Efficient implementation of lazy narrowing into PROLOG. In *LOPSTR'92*, 1993. Previous version: Some Techniques for the Efficient Implementation of Lazy Narrowing, Technical Report -FIM.75/LyS/92, Facultad de Informatica, Universidad Politecnica de Madrid, 1992.

[27] J. R. Kennaway. Sequential evaluation strategies for parallel-or and related reduction systems. *Annals of Pure and Applied Logic*, 43:31–56, 1989.

[28] J. R. Kennaway. The specificity rule for lazy pattern-matching in ambiguous term rewrite systems. In *Third European Symp. on Programming*, pages 256–270, 1990. *LNCS* 432.

[29] 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. Previous version: Term rewriting systems, Technical Report CS-R9073, Stichting Mathematisch Centrum, Amsterdam, 1990.

[30] J. W. Klop and A. Middeldorp. Sequentiality in orthogonal term rewriting systems. *Journal of Symbolic Computation*, pages 161–195, 1991. Previous version: Technical Report CS-R8932, Stichting Mathematisch Centrum, Amsterdam, The Netherlands, 1989.

[31] S. Krischer and A. Bockmayr. Detecting redundant narrowing derivations by the LSE-SL reducibility test. In *Proc. RTA'91.* Springer LNCS 488, 1991.

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

[33] L. Maranget. Optimal derivation in weak lambda-calculi and in orthogonal terms rewriting systems. In *17th Annual Symp. on Principles of Prog. Languages*, pages 255–269. ACM, 1990.

[34] A. Middeldorp, August 1993. Personal Communication.

[35] A. Middeldorp and E. Hamoen. Counterexamples to completeness results for basic narrowing (extended abstract). In *Proceedings of the Third International Conference on Algebraic and Logic Programming*, pages 244–258, Volterra, Italy, September 1992.

[36] 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*, pages 298–317. Springer LNCS 463, 1990.

[37] 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.

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

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

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

[41] M. J. Oyamaguchi. Nv-sequentiality: A decidable condition for call-by-need computations in term rewriting systems. *SIAM Journal on Computation*, 22(1):114–135, 1993.

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

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

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

[45] R. C. Sekar and I. V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 230–241, Philadelphia, PA, June 1990.

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

[47] 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.

[48] S. Thatte. A refinement of strong sequentiality for term rewriting with constructors. *Information and Computation*, 72:46–65, 1987.

[49] J.-H. You. Enumerating outer narrowing derivations for constructor-based term rewriting systems. *Journal of Symbolic Computation*, 7:319–341, 1989.