

# Implementing Natural Rewriting and Narrowing Efficiently<sup>\*</sup>

Santiago Escobar

DSIC, Universidad Politécnica de Valencia  
Camino de Vera, s/n, E-46022 Valencia, Spain  
`sescobar@dsic.upv.es`

**Abstract.** *Outermost-needed rewriting/narrowing* is a sound and complete optimal demand-driven strategy for the class of inductively sequential constructor systems. Its parallel extension, known as *weakly*, deals with non-inductively sequential constructor systems. Recently, refinements of (weakly) outermost-needed rewriting and narrowing have been obtained. These new strategies are called *natural rewriting* and *natural narrowing*, respectively, and incorporate a better treatment of demandedness. In this paper, we address the problem of how to implement natural rewriting and narrowing efficiently by using a refinement of the notion of definitional tree, which we call matching definitional tree. We also show how to compile natural rewriting and narrowing to Prolog and provide some promising experimental results.

## 1 Introduction

A challenging problem in modern programming languages is the discovery of sound and complete evaluation strategies which are ‘optimal’ w.r.t. some efficiency criterion (typically the number of evaluation steps and the avoidance of infinite, failing or redundant derivations) and which are easily implementable.

A sound and complete rewrite strategy for the class of inductively sequential constructor systems (CSs) is *outermost-needed rewriting* [3]. The extension to narrowing is called *outermost-needed narrowing* (or *needed narrowing*) [5]. Intuitively, a left-linear CS is inductively sequential if there exists some branching selection structure that is inherent to the rules. The optimality properties associated to inductively sequential CSs explain why outermost-needed narrowing has become useful in functional logic programming as the functional logic counterpart of Huet and Lévy’s strongly needed reduction [13]. *Weakly outermost-needed rewriting* [4] is defined for non-inductively sequential CSs. Its extension to narrowing is called *weakly outermost-needed narrowing* [4] and is considered as the functional logic counterpart of Sekar and Ramakrishnan’s parallel needed reduction [15].

Whereas outermost-needed rewriting and narrowing are optimal w.r.t. to inductively sequential CSs, weakly outermost-needed rewriting and narrowing

---

<sup>\*</sup> Work partially supported by MCyT under grants TIC2001-2705-C03-01, HA2001-0059 and HU2001-0019.

do not work appropriately on non-inductively sequential CSs, as shown by the following example.

*Example 1.* Consider Berry's program [15] where  $T$  and  $F$  are constructor symbols and  $X$  is a variable:

$$B(T, F, X) = T \qquad B(F, X, T) = T \qquad B(X, T, F) = T$$

This CS is not inductively sequential since there is no branching selection structure in the rules. However, although the CS is not inductively sequential, some terms can still be reduced sequentially, where 'to be reduced sequentially' is understood as the property of reducing only positions that are unavoidable (or "needed") when attempting to obtain a normal form (see [13]). For instance, the term  $B(B(T, F, T), B(F, T, T), F)$  has a unique 'optimal' rewrite sequence which achieves its associated normal form  $T$

$$B(B(T, F, T), B(F, T, T), F) \rightarrow B(B(T, F, T), T, F) \rightarrow T$$

However, weakly outermost-needed rewriting is not optimal since, besides the previous optimal sequence, the sequence

$$B(B(T, F, T), B(F, T, T), F) \rightarrow B(T, B(F, T, T), F) \rightarrow B(T, T, F) \rightarrow T$$

is also obtained. The reason is that weakly outermost-needed rewriting partitions the CS into the inductively sequential subsets  $\mathcal{R}_1 = \{B(X, T, F) = T\}$  and  $\mathcal{R}_2 = \{B(T, F, X) = T, B(F, X, T) = T\}$  in such a way that the first step of the former (optimal) rewriting sequence is obtained w.r.t. subset  $\mathcal{R}_1$  whereas the first (useless) step of the latter rewriting sequence is obtained w.r.t. subset  $\mathcal{R}_2$ . Note that the problem also occurs in weakly outermost-needed narrowing. For instance, term  $B(X, B(F, T, T), F)$  has the optimal narrowing sequence

$$B(X, B(F, T, T), F) \rightsquigarrow_{id} B(X, T, F) \rightsquigarrow_{id} T$$

whereas weakly outermost-needed narrowing also produces the following non-optimal (due to the unnecessary substitution  $\{X \mapsto T\}$ ) narrowing sequence

$$B(X, B(F, T, T), F) \rightsquigarrow_{\{X \mapsto T\}} B(T, T, F) \rightsquigarrow_{id} T$$

On the other hand, outermost-needed rewriting and narrowing are optimal for inductively sequential CSs only when non-failing input terms are considered, i.e. terms which can be reduced or narrowed to a constructor head-normal form. Hence, some refinement is still possible for failing input terms.

Modern (multiparadigm) programming languages apply computational strategies which are based on some notion of demandness of a position in a term by a rule (see [7]). Programs in these languages are commonly modeled by left-linear CSs, and computational strategies take advantage of this constructor condition (see [2,14]).

*Example 2.* Consider the following TRS borrowed from [8] defining the symbol  $\div$ , which encodes the division function between natural numbers.

$$\begin{array}{ll} 0 \div s(N) = 0 & M - 0 = M \\ s(M) \div s(N) = s((M-N) \div s(N)) & s(M) - s(N) = M-N \end{array}$$

Consider the term  $t = 10! \div 0$ , which is a (non-constructor) head-normal form. Outermost-needed rewriting forces<sup>1</sup> the reduction of the first argument and evaluates  $10!$ , which is useless. The reason is that outermost-needed rewriting uses a data structure called *definitional tree* which encodes the branching selection structure existing in the rules without testing whether the rules associated to each branch could ever be matched to the term or not. A similar problem occurs when narrowing the term  $X \div 0$ , since variable  $X$  is instantiated to  $0$  or  $s$ . However, neither instantiation is really necessary.

In [9], we proposed a solution to these two problems, namely the non-optimal evaluation for non-inductively sequential programs and the unnecessary evaluation in failing terms, which is based on a suitable extension of the demandedness notion associated to weakly outermost-needed rewriting and narrowing. The new strategies are called *natural rewriting* and *natural narrowing*, respectively. Our strategies incorporate a better treatment of demandedness and enjoy good computational properties; in particular, we show how to use them for computing (head-)normal forms and we prove they are conservative w.r.t. (weakly) outermost-needed rewriting and (weakly) outermost-needed narrowing. Moreover, we defined a new class of CSs called *inductively sequential preserving* where natural rewriting and narrowing preserve optimality for sequential parts of the program. This new class of CSs is based on the extension of the notion of inductive sequentiality from defined function symbols to terms and is larger than the class of inductively sequential CSs.

In this paper, we address how to implement natural rewriting and natural narrowing efficiently. After some preliminaries in Section 2, in Section 3, we provide a generalization of the notion of definitional tree, which we call *matching definitional tree*. In Section 4, we present how to reproduce natural rewriting and natural narrowing by traversing matching definitional trees. In Section 5, we show that it is possible to implement natural rewriting and narrowing efficiently by compiling to Prolog. Finally, Section 6 presents our conclusions.

## 2 Preliminaries

We assume some familiarity with term rewriting (see [16] for missing definitions) and narrowing (see [10] for missing definitions). Let  $R \subseteq A \times A$  be a binary relation on a set  $A$ . We denote the reflexive closure of  $R$  by  $R^=$ , its transitive closure by  $R^+$ , and its reflexive and transitive closure by  $R^*$ . An element  $a \in A$  is an  $R$ -normal form, if there exists no  $b$  such that  $a R b$ . We say that  $b$  is an  $R$ -normal form of  $a$  (written  $a R^! b$ ), if  $b$  is an  $R$ -normal form and  $a R^* b$ .

Throughout the paper,  $\mathcal{X}$  denotes a countable set of variables  $\{x, y, \dots\}$  and  $\mathcal{F}$  denotes a *many-sorted* signature, i.e. a set of function symbols  $\{f, g, \dots\}$  grouped into sorts. We denote the set of terms built from  $\mathcal{F}$  and  $\mathcal{X}$  by  $T(\mathcal{F}, \mathcal{X})$ . A  $k$ -tuple  $t_1, \dots, t_k$  of terms is written  $\bar{t}$ . A term is said to be linear if it has

<sup>1</sup> Note that this behavior is independent of the fact that two possible definitional trees exist for symbol  $\div$  (see [9, Example 21]).

no multiple occurrences of a single variable. Let  $Subst(\mathcal{T}(\mathcal{F}, \mathcal{X}))$  denote the set of substitutions. We denote by  $id$  the “identity” substitution:  $id(x) = x$  for all  $x \in \mathcal{X}$ . Terms are ordered by the preorder  $\leq$  of “relative generality”, i.e.  $s \leq t$  if there exists  $\sigma$  s.t.  $\sigma(s) = t$ . Term  $t$  is a variant of  $s$  if  $t \leq s$  and  $s \leq t$ . A *most general unifier (mgu)* of  $t, s$  is a unifier  $\sigma$  such that for each unifier  $\sigma'$  of  $t, s$  there exists  $\theta$  such that  $\sigma' = \theta \circ \sigma$ .

By  $\mathcal{Pos}(t)$  we denote the set of positions of a term  $t$ . Given a set  $S \subseteq \mathcal{F} \cup \mathcal{X}$ ,  $\mathcal{Pos}_S(t)$  denotes positions in  $t$  where symbols in  $S$  occur. We denote the root position by  $\Lambda$ . Given positions  $p, q$ , we denote its concatenation as  $p.q$ . Positions are ordered by the standard prefix ordering  $\leq$ . The subterm at position  $p$  of  $t$  is denoted as  $t|_p$ , and  $t[s]_p$  is the term  $t$  with the subterm at position  $p$  replaced by  $s$ . The symbol labeling the root of  $t$  is denoted as  $root(t)$ .

A rewrite rule is an ordered pair  $(l, r)$ , written  $l \rightarrow r$ , with  $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  and  $l \notin \mathcal{X}$ . The left-hand side (*lhs*) of the rule is  $l$  and the right-hand side (*rhs*) of the rule is  $r$ . A TRS is a pair  $\mathcal{R} = (\mathcal{F}, R)$  where  $R$  is a set of rewrite rules.  $L(\mathcal{R})$  denotes the set of *lhs*'s of  $\mathcal{R}$ . A TRS  $\mathcal{R}$  is left-linear if for all  $l \in L(\mathcal{R})$ ,  $l$  is a linear term. Given  $\mathcal{R} = (\mathcal{F}, R)$ , we take  $\mathcal{F}$  as the disjoint union  $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$  of symbols  $c \in \mathcal{C}$ , called *constructors* and symbols  $f \in \mathcal{D}$ , called *defined functions*, where  $\mathcal{D} = \{root(l) \mid l \rightarrow r \in R\}$  and  $\mathcal{C} = \mathcal{F} - \mathcal{D}$ . A pattern is a term  $f(l_1, \dots, l_k)$  where  $f \in \mathcal{D}$  and  $l_i \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ , for  $1 \leq i \leq k$ . A TRS  $\mathcal{R} = (\mathcal{C} \uplus \mathcal{D}, R)$  is a constructor system (CS) if all *lhs*'s are patterns. A term  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  rewrites to  $s$  (at position  $p$ ), written  $t \xrightarrow{p}_{l \rightarrow r} s$  (or just  $t \rightarrow s$ ), if  $t|_p = \sigma(l)$  and  $s = t[\sigma(r)]_p$ , for some rule  $l \rightarrow r \in R$ ,  $p \in \mathcal{Pos}(t)$  and substitution  $\sigma$ . The subterm  $\sigma(l)$  in  $t$  is called a *redex*. On the other hand, a term  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  narrows to  $s$  (at position  $p$  with substitution  $\sigma$ ), written  $t \rightsquigarrow_{\{p, \sigma, l \rightarrow r\}} s$  (or just  $t \rightsquigarrow_\sigma s$ ) if  $p$  is a non-variable position in  $t$  and  $\sigma(t) \xrightarrow{p}_{l \rightarrow r} s$ . A term  $t$  is a *head-normal form* (or *root-stable*) if it cannot be reduced to a redex.

### 3 Matching Definitional Trees

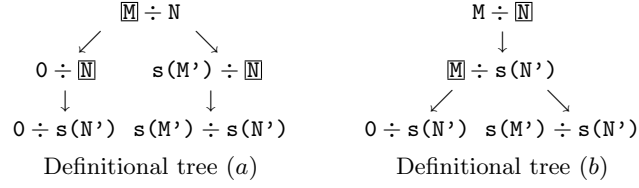
In order to efficiently implement natural rewriting and narrowing, we must integrate the demandedness notion of natural rewriting and narrowing [9] into a statically built structure, as happens in weakly outermost-needed rewriting and narrowing which use definitional trees [3]. Thus, we define *matching definitional trees*. First, we recall the definition of a (generalized) definitional tree.

**Definition 1.** [4]  $\mathcal{T}$  is a generalized definitional tree, or *gdt*, with pattern  $\pi$  iff one of the following cases holds:

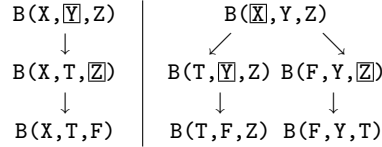
$\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \dots, \mathcal{T}_k)$  where  $\pi$  is a pattern,  $o$  is the occurrence (called *inductive*) of a variable of  $\pi$ , the sort of  $\pi|_o$  has different constructors  $c_1, \dots, c_k$  for  $k > 0$ , and for all  $i$  in  $\{1, \dots, k\}$ ,  $\mathcal{T}_i$  is a *gdt* with pattern  $\pi[c_i(\bar{x})]_o$ , where  $\bar{x}$  are new distinct variables.

$\mathcal{T} = leaf(\pi, l \rightarrow r)$  where  $\pi$  is a pattern and  $l \rightarrow r$  is a rule such that  $\pi$  and  $l$  are variants.

$\mathcal{T} = or(\mathcal{T}_1, \dots, \mathcal{T}_k)$  where  $k > 1$  and each  $\mathcal{T}_i$  is a *gdt* with pattern  $\pi$ .



**Fig. 1.** The two possible definitional trees for the symbol  $\div$



**Fig. 2.** A partition of definitional trees for the symbol  $B$

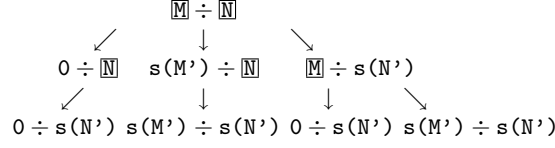
A *definitional tree* is a generalized definitional tree without *or*-nodes. A *parallel definitional tree* is a generalized definitional tree where only one *or*-node is allowed at the top of the tree. A defined symbol  $f$  is called *inductively sequential* if there exists a definitional tree  $\mathcal{T}$  with pattern  $f(x_1, \dots, x_k)$  (where  $x_1, \dots, x_k$  are different variables) whose leaves contain all and only the rules defining  $f$ . In this case, we say that  $\mathcal{T}$  is a definitional tree for  $f$ , denoted as  $\mathcal{T}_f$ . A left-linear CS  $\mathcal{R}$  is *inductively sequential* if all its defined function symbols are inductively sequential. It is often convenient and simplifies understanding to provide a graphical representation of definitional trees as a tree of patterns where the inductive position in *branch*-nodes is surrounded by a box [3].

*Example 3.* The symbol  $\div$  in Example 2 is inductively sequential since there exists a definitional tree for pattern  $M \div N$ . Figure 1 shows the two possible definitional trees.

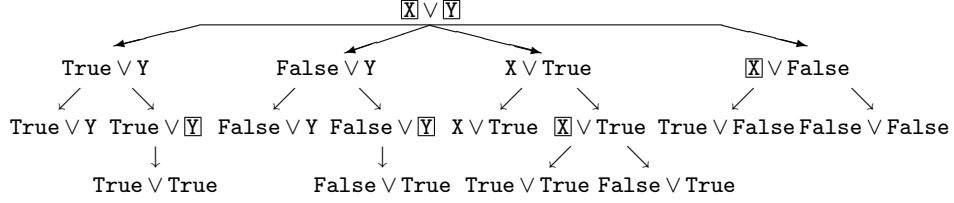
When a function symbol is not inductively sequential, a parallel definitional tree, instead of a definitional tree, is obtained. The graphical representation of a parallel definitional tree corresponds to a set of definitional trees.

*Example 4.* The symbol  $B$  in Example 1 is non-inductively sequential, since a rule partition that splits its rules into subsets that are inductively sequential is necessary. Figure 2 shows a parallel definitional tree for symbol  $B$ .

Definitional trees are used by (weakly) outermost-needed rewriting and narrowing as a finite state automaton to compute the demanded positions to be reduced (in line with [13,15]). However, definitional trees merge two different processes: the pattern matching process and the evaluation process through demanded positions. These two processes coincide in the case of inductively sequential and non-failing terms because the evaluation sequence corresponds to the pattern matching sequence, but do not coincide for non-inductively sequential or failing terms where the pattern matching process may determine a (possibly better) evaluation sequence (as shown in Examples 1 and 2). We define a



**Fig. 3.** A matching definitional tree for symbol  $\div$



**Fig. 4.** A matching definitional tree for symbol  $\vee$

*matching definitional tree* as a definitional tree where more than one inductive position is allowed for *branch*-nodes.

**Definition 2.**  $\mathcal{T}$  is a matching definitional tree, or mdt, with pattern  $\pi$  iff one of the following cases holds:

$\mathcal{T} = \text{branch}(\pi, (o_1, \mathcal{T}_1^1, \dots, \mathcal{T}_{k_1}^1), \dots, (o_n, \mathcal{T}_1^n, \dots, \mathcal{T}_{k_n}^n))$  where  $\pi$  is a pattern,  $o_1, \dots, o_n$  with  $n > 0$  are occurrences of variables of  $\pi$ , the sort of  $\pi|_{o_i}$  has different constructors  $c_1^i, \dots, c_{k_i}^i$  for  $i$  in  $\{1, \dots, n\}$  and  $k_i > 0$ , and for all  $j$  in  $\{1, \dots, k_i\}$ ,  $\mathcal{T}_j^i$  is a mdt with pattern  $\pi[c_j^i(\bar{x})]_{o_i}$ , where  $\bar{x}$  are new distinct variables.

$\mathcal{T} = \text{leaf}(\pi, l \rightarrow r)$  where  $\pi$  is a pattern and  $l \rightarrow r$  is a rule such that  $l \leq \pi$ .

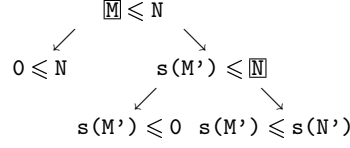
$\mathcal{T} = \text{or}(\mathcal{T}_1, \dots, \mathcal{T}_k)$  where  $k > 1$  and each  $\mathcal{T}_i$  is a mdt with pattern  $\pi$ .

For each defined symbol  $f$  in a left-linear CS, we can build a matching definitional tree  $\mathcal{T}$  with pattern  $f(x_1, \dots, x_k)$  (where  $x_1, \dots, x_k$  are different variables) whose leaves contain all and only the rules defining  $f$ . The graphical representation is similar to that of a definitional tree with the difference that it is possible to have more than one inductive position in *branch*-nodes and *or*-nodes are identified as nodes which do not have any boxed position. We denote *branch*-nodes that have only one position  $o$  as simply  $\text{branch}(\pi, o, \mathcal{T}_1, \dots, \mathcal{T}_k)$  (as happens in gdt's).

*Example 5.* Figure 3 represents the following<sup>2</sup> matching definitional tree which could be associated to the symbol  $\div$  in Example 2:

$$\begin{aligned} &\text{branch}(\mathbb{M} \div \mathbb{N}, (1, \text{branch}(0 \div \mathbb{N}, 2, \text{leaf}(0 \div \mathbf{s}(\mathbb{N}')))), \\ &\quad \text{branch}(\mathbf{s}(\mathbb{M}') \div \mathbb{N}, 2, \text{leaf}(\mathbf{s}(\mathbb{M}') \div \mathbf{s}(\mathbb{N}'))), \\ &\quad (2, \text{branch}(\mathbb{M} \div \mathbf{s}(\mathbb{N}'), 1, \text{leaf}(0 \div \mathbf{s}(\mathbb{N}'))), \\ &\quad \quad \text{leaf}(\mathbf{s}(\mathbb{M}') \div \mathbf{s}(\mathbb{N}')))) \end{aligned}$$

<sup>2</sup> In this paper, we often omit the rule in a leaf node for simplicity.



**Fig. 5.** The definitional tree and matching definitional tree for symbol  $\leq$

*Example 6.* Consider the following parallel-or TRS from [9]:

$$\text{True} \vee X = \text{True} \quad X \vee \text{True} = \text{True} \quad \text{False} \vee X = X$$

Figure 4 denotes the following *mdt* associated to the symbol  $\vee$ :

$$\begin{aligned}
& \text{branch}(X \vee Y, (1, \text{or}(\text{leaf}(\text{True} \vee Y), \\
& \quad \text{branch}(\text{True} \vee Y, 2, \text{leaf}(\text{True} \vee \text{True}))), \\
& \quad \text{or}(\text{leaf}(\text{False} \vee Y), \\
& \quad \quad \text{branch}(\text{False} \vee Y, 2, \text{leaf}(\text{False} \vee \text{True}))), \\
& (2, \text{or}(\text{leaf}(X \vee \text{True}), \\
& \quad \text{branch}(X \vee \text{True}, 1, \text{leaf}(\text{True} \vee \text{True}), \\
& \quad \quad \text{leaf}(\text{False} \vee \text{True}))), \\
& \quad \text{branch}(X \vee \text{False}, 1, \text{leaf}(\text{True} \vee \text{False}), \\
& \quad \quad \text{leaf}(\text{False} \vee \text{False}))),
\end{aligned}$$

Note that every definitional tree is also a matching definitional tree.

*Example 7.* Consider the following defined symbol  $\leq$ , which is an example commonly used to refer to definitional trees [5]:

$$0 \leq N = \text{True} \quad s(M) \leq 0 = \text{False} \quad s(M) \leq s(N) = M \leq N$$

Since this symbol has only one possible definitional tree (instead of symbol  $\div$  of Example 2 which has two), the associated definitional tree of Figure 5 corresponds to its matching definitional tree:

$$\begin{aligned}
& \text{branch}(M \leq N, 1, \text{leaf}(0 \leq N), \\
& \quad \text{branch}(s(M') \leq N, 2, \text{leaf}(s(M') \leq 0), \text{leaf}(s(M') \leq s(N'))))
\end{aligned}$$

## 4 Evaluation through matching definitional trees

In inductively sequential left-linear CSs, the order of evaluation is determined by a mapping  $\varphi$  which implements the strategy by traversing definitional trees as a finite state automaton, in order to compute the demanded positions to be reduced [3].

In order to implement natural rewriting efficiently, we define a mapping  $\text{mt}$  which traverses a matching definitional tree as a pattern matching finite state automaton and returns the set of demanded positions associated to a concrete *branch*-node if all its inductive positions are rooted by non-constructor symbols.

**Definition 3.** The function  $\mathbf{mt}$  takes two arguments: an operation-rooted term,  $t$ , and a mdt,  $\mathcal{T}$ , such that  $\mathit{pattern}(\mathcal{T})$  and  $t$  unify. The function  $\mathbf{mt}$  yields a set of tuples of the form  $(p, R)$ , where  $p$  is a position of  $t$ , and  $R$  is a rule  $l \rightarrow r$  of  $\mathcal{R}$ . The function  $\mathbf{mt}$  is defined as follows:

$$\mathbf{mt}(t, \mathcal{T}) \ni \left\{ \begin{array}{l} (\Lambda, R) \quad \text{if } \mathcal{T} = \mathit{leaf}(\pi, R); \\ (p, R) \quad \text{if } \mathcal{T} = \mathit{or}(\mathcal{T}_1, \dots, \mathcal{T}_k) \text{ and } (p, R) \in \mathbf{mt}(t, \mathcal{T}_i) \text{ for some } i; \\ (p, R) \quad \text{if } \mathcal{T} = \mathit{branch}(\pi, (o_1, \mathcal{T}_1^1, \dots, \mathcal{T}_{k_1}^1), \dots, (o_n, \mathcal{T}_1^n, \dots, \mathcal{T}_{k_n}^n)), \\ \quad \text{root}(t|_{o_i}) \in \mathcal{C} \text{ for some } 1 \leq i \leq n, \mathit{pattern}(\mathcal{T}_j^i) \leq t \\ \quad \text{for some } 1 \leq j \leq k_i, \text{ and } (p, R) \in \mathbf{mt}(t, \mathcal{T}_j^i); \\ (o_i.p, R) \text{ if } \mathcal{T} = \mathit{branch}(\pi, (o_1, \mathcal{T}_1^1, \dots, \mathcal{T}_{k_1}^1), \dots, (o_n, \mathcal{T}_1^n, \dots, \mathcal{T}_{k_n}^n)), \\ \quad \text{root}(t|_{o_i}) \in \mathcal{D} \text{ for all } 1 \leq i \leq n, \\ \quad \text{and } (p, R) \in \mathbf{mt}(t|_{o_i}, \mathcal{T}_{\mathit{root}(t|_{o_i})}) \text{ for some } 1 \leq i \leq n. \end{array} \right.$$

*Example 8.* Consider the TRS and term  $t = 10! \div 0$  in Example 2 together with the matching definitional tree  $\mathcal{T}_{\div}$  in Example 5 (and Figure 3). We have  $\mathbf{mt}(t, \mathcal{T}_{\div}) = \emptyset$  since the two inductive positions of the topmost *branch*-node are not operation-rooted and no subtree can be selected: the two first subtrees because position 1 is operation-rooted and the third one because it needs an *s* constructor symbol at position 2, instead of 0.

Outermost-needed narrowing is determined by a mapping  $\lambda$  which implements the strategy by traversing definitional trees [5]. Here, we define a mapping  $\mathbf{mnt}$  which extends  $\mathbf{mt}$  to narrowing as  $\lambda$  extends  $\varphi$ .

**Definition 4.** The function  $\mathbf{mnt}$  takes two arguments: an operation-rooted term,  $t$ , and a mdt,  $\mathcal{T}$ , such that  $\mathit{pattern}(\mathcal{T})$  and  $t$  unify. The function  $\mathbf{mnt}$  yields a set of triples of the form  $(p, R, \sigma)$ , where  $p$  is a position of  $t$ ,  $R$  is a rule  $l \rightarrow r$  of  $\mathcal{R}$ , and  $\sigma$  is a unifier of  $\mathit{pattern}(\mathcal{T})$  and  $t$ . The function  $\mathbf{mnt}$  is defined as follows:

$$\mathbf{mnt}(t, \mathcal{T}) \ni \left\{ \begin{array}{l} (\Lambda, R, \sigma) \quad \text{if } \mathcal{T} = \mathit{leaf}(\pi, R) \text{ and } \sigma = \mathit{mgu}(\pi, t); \\ (p, R, \sigma) \quad \text{if } \mathcal{T} = \mathit{or}(\mathcal{T}_1, \dots, \mathcal{T}_k) \text{ and } (p, R, \sigma) \in \mathbf{mnt}(t, \mathcal{T}_i) \text{ for some } i; \\ (p, R, \sigma) \quad \text{if } \mathcal{T} = \mathit{branch}(\pi, (o_1, \mathcal{T}_1^1, \dots, \mathcal{T}_{k_1}^1), \dots, (o_n, \mathcal{T}_1^n, \dots, \mathcal{T}_{k_n}^n)), \\ \quad \text{root}(t|_{o_i}) \in \mathcal{C} \text{ for some } 1 \leq i \leq n, \mathit{pattern}(\mathcal{T}_j^i) \leq t \\ \quad \text{for some } 1 \leq j \leq k_i, \text{ and } (p, R, \sigma) \in \mathbf{mnt}(t, \mathcal{T}_j^i); \\ (o_i.p, R, \sigma) \text{ if } \mathcal{T} = \mathit{branch}(\pi, (o_1, \mathcal{T}_1^1, \dots, \mathcal{T}_{k_1}^1), \dots, (o_n, \mathcal{T}_1^n, \dots, \mathcal{T}_{k_n}^n)), \\ \quad \text{root}(t|_{o_i}) \notin \mathcal{C} \text{ for all } 1 \leq i \leq n, \text{root}(t|_{o_i}) \in \mathcal{D} \text{ for} \\ \quad \text{some } 1 \leq i \leq n, \text{ and } (p, R, \sigma) \in \mathbf{mnt}(t|_{o_i}, \mathcal{T}_{\mathit{root}(t|_{o_i})}); \\ (p, R, \theta \circ \sigma) \text{ if } \mathcal{T} = \mathit{branch}(\pi, (o_1, \mathcal{T}_1^1, \dots, \mathcal{T}_{k_1}^1), \dots, (o_n, \mathcal{T}_1^n, \dots, \mathcal{T}_{k_n}^n)), \\ \quad \text{root}(t|_{o_i}) \notin \mathcal{C} \text{ for all } 1 \leq i \leq n, t|_{o_i} \in \mathcal{X} \text{ for some} \\ \quad 1 \leq i \leq n, \theta = \mathit{mgu}(t, \mathit{pattern}(\mathcal{T}_j^i)) \text{ for some} \\ \quad 1 \leq j \leq k_i, \text{ and } (p, R, \sigma) \in \mathbf{mnt}(\theta(t), \mathcal{T}_j^i). \end{array} \right.$$



rewriting strategies select ‘popular’ demanded positions over the available set. In [9], we formalized a notion of popularity of demanded positions which makes the difference by counting the number of times a position is demanded by some rule, and then selects the most (frequently) demanded positions covering all eventually applicable rules.

**Definition 7.** [9] We define the multiset of demanded positions of a term  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  w.r.t. TRS  $\mathcal{R}$  as  $DP_{\mathcal{R}}(t) = \oplus\{DP_l(t) \mid l \rightarrow r \in \mathcal{R} \wedge \text{root}(t) = \text{root}(l)\}$  where  $M_1 \oplus M_2$  is the union of multisets  $M_1$  and  $M_2$ .

*Example 11.* Continuing Example 10, we have  $DP_{\mathcal{R}}(t) = DP_{\mathcal{R}}(t') = \{1, 2, 2\}$ .

**Definition 8 (Demanded positions).** [9] We define the set of demanded positions of a term  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  w.r.t. TRS  $\mathcal{R}$  as

$$DP_{\mathcal{R}}^m(t) = \{p \in DP_{\mathcal{R}}(t) \mid \exists l \in L(\mathcal{R}). p \in DP_l(t) \text{ and} \\ \forall q \in DP_{\mathcal{R}}(t) : p <_{DP_{\mathcal{R}}(t)} q \Rightarrow l|_q \in \mathcal{X}\}$$

where  $x <_M y$  denotes that the number of occurrences of  $x$  in the multiset  $M$  is less than the number of occurrences of  $y$ .

*Example 12.* Continuing Example 11, we have  $DP_{\mathcal{R}}^m(t) = DP_{\mathcal{R}}^m(t') = \{2\}$  since even though position 1 is demanded, the redex at position 2 is the most frequently demanded.

*Example 13.* Consider the TRS  $\mathcal{R}$  of Example 6 and the term  $t = (\text{True} \vee \text{False}) \vee (\text{True} \vee \text{False})$ . We have  $DP_{\mathcal{R}}^m(t) = \{1, 2\}$  since position 1 alone does not cover the rule  $X \vee \text{True} = \text{True}$ .

The following definition establishes the strategy used in natural rewriting for selecting demanded positions.

**Definition 9 (Natural rewriting).** [9] Given a term  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  and a TRS  $\mathcal{R}$ ,  $\mathfrak{m}(t)$  is defined as the smallest set satisfying

$$\mathfrak{m}(t) \ni \begin{cases} (A, l \rightarrow r) & \text{if } l \rightarrow r \in \mathcal{R} \text{ and } l \leq t \\ (p.q, l \rightarrow r) & \text{if } p \in DP_{\mathcal{R}}^m(t) \text{ and } (q, l \rightarrow r) \in \mathfrak{m}(t|_p) \end{cases}$$

We consider  $\mathfrak{m}$  simply as a function returning positions if the rules selected for reduction are not relevant or they are clear from the context. We say term  $t$  reduces by *natural rewriting* to term  $s$ , denoted by  $t \xrightarrow{\mathfrak{m}}_{\{p, l \rightarrow r\}} s$  (or simply  $t \xrightarrow{\mathfrak{m}} s$ ) if  $(p, l \rightarrow r) \in \mathfrak{m}(t)$ .

*Example 14.* Continuing Example 12, we have  $\mathfrak{m}(t) = \{2\}$  and the only possible natural rewriting step is:  $\underline{B(B(T, F, T), B(F, T, T), F)} \xrightarrow{\mathfrak{m}} \underline{B(B(T, F, T), T, F)}$

*Example 15.* Continuing Example 13, it is clear that  $\mathfrak{m}(t) = \{1, 2\}$  and, thus, there exist two possible natural rewriting steps:

$$\begin{aligned} \underline{(\text{True} \vee \text{False})} \vee (\text{True} \vee \text{False}) &\xrightarrow{\mathfrak{m}} \text{True} \vee (\text{True} \vee \text{False}) \\ (\text{True} \vee \text{False}) \vee \underline{(\text{True} \vee \text{False})} &\xrightarrow{\mathfrak{m}} (\text{True} \vee \text{False}) \vee \text{True} \end{aligned}$$

Now, we prove that natural rewriting is appropriately reproduced by using matching definitional trees and Definition 3. First, we give a class of matching definitional trees, called *safe matching definitional trees*, where the inductive positions of a *branch*-node correspond to the demanded positions obtained by the operator  $DP_{\mathcal{R}}^m$ . In the following, we denote the set of rules appearing at leaves of a tree as  $leaves(\mathcal{T})$ . Similarly, given a TRS  $\mathcal{R}$ , we denote the set of rules which are instantiations of a pattern  $\pi$  as  $rules_{\mathcal{R}}(\pi) = \{l \rightarrow r \in \mathcal{R} \mid \pi \leq l\}$ . A *mdt*  $\mathcal{T}$  is called *total* if it contains all and only the rules eventually applicable to the pattern of  $\mathcal{T}$  (i.e.  $rules_{\mathcal{R}}(pattern(\mathcal{T})) = leaves(\mathcal{T})$ ).

**Definition 10.** *Let  $\mathcal{R} = (\mathcal{F}, R)$  be a left-linear CS and  $\mathcal{T}$  be a matching definitional tree for symbol  $f \in \mathcal{D}$  with pattern  $\pi$ . We say  $\mathcal{T}$  is a safe matching definitional tree for symbol  $f$  iff one of the following cases holds:*

$$\begin{aligned} \mathcal{T} &= branch(\pi, (o_1, \mathcal{T}_1^1, \dots, \mathcal{T}_{k_1}^1), \dots, (o_n, \mathcal{T}_1^n, \dots, \mathcal{T}_{k_n}^n)), DP_{\mathcal{R}}^m(\pi) = \{o_1, \dots, o_n\}, \\ &\quad \text{and } \mathcal{T}_1^1, \dots, \mathcal{T}_{k_1}^1, \dots, \mathcal{T}_1^n, \dots, \mathcal{T}_{k_n}^n \text{ are total, safe mdt's.} \\ \mathcal{T} &= leaf(\pi, l \rightarrow r) \text{ and } l \rightarrow r \in \mathcal{R}. \\ \mathcal{T} &= or(\mathcal{T}_1, \dots, \mathcal{T}_k), \mathcal{T}_1, \dots, \mathcal{T}_k \text{ are safe mdt's, } \mathcal{T}_1, \dots, \mathcal{T}_{k-1} \text{ are leaves with pat-} \\ &\quad \text{tern } \pi, \text{ and each rule } l \rightarrow r \in leaves(\mathcal{T}) \text{ is unique among } \mathcal{T}_1, \dots, \mathcal{T}_k. \end{aligned}$$

Note that the matching definitional trees of Figures 3, 4, 5 and 6 are safe *mdt*'s whereas the two definitional trees of Figure 1 are not. Indeed, it is worth noting that the construction of safe *mdt*'s is easily achievable by using the function  $DP_{\mathcal{R}}^m$  for *branch*-nodes and creating *or*-nodes when a pattern is a redex and also has demanded positions. Furthermore, there is only one safe *mdt* per function symbol, due to the use of  $DP_{\mathcal{R}}^m$ . Now, we prove that both definitions of natural rewriting coincide for safe *mdt*'s.

**Theorem 1.** *Let  $\mathcal{R} = (\mathcal{F}, R)$  be a left-linear CS,  $f \in \mathcal{D}$ ,  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  s.t.  $root(t) = f$ , and  $\mathcal{T}_f$  be a safe *mdt* for  $f$ . Then,  $mt(t, \mathcal{T}_f) = m(t)$ .*

## 4.2 Natural Narrowing

The following definition establishes the strategy used in natural narrowing for allowing narrowing steps.

**Definition 11 (Natural narrowing).** [9] *Given a term  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  and a TRS  $\mathcal{R}$ ,  $mn(t)$  is defined<sup>3</sup> as the smallest set satisfying*

$$mn(t) \ni \begin{cases} (\Lambda, id, l \rightarrow r) \text{ if } l \rightarrow r \in \mathcal{R} \text{ and } l \leq t \\ (p.q, \theta, l \rightarrow r) \text{ if } p \in DP_{\mathcal{R}}^m \cap Pos_{\mathcal{D}}(t) \text{ and } (q, \theta, l \rightarrow r) \in mn(t|_p) \\ (p, \theta \circ \sigma, l \rightarrow r) \text{ if } p \in DP_{\mathcal{R}}^m.t|_p = x \in \mathcal{X}, c \text{ is in the sort of } x, \sigma(x) = c(\bar{w}), \\ \quad \bar{w} \text{ are fresh variables, and } (p, \theta, l \rightarrow r) \in mn(\sigma(t)) \end{cases}$$

<sup>3</sup> This definition is a simplification of that of [9] and differs only in the selection of demanded positions rooted by defined symbols. Note that both definitions satisfy the same properties described in [9].

We say term  $t$  narrows by *natural narrowing* to term  $s$  at position  $p$  using substitution  $\sigma$ , denoted by  $t \xrightarrow[\{p, l \rightarrow r, \sigma\}]{m} s$  (or simply  $t \xrightarrow[\sigma]{m} s$ ), if  $(p, \sigma, l \rightarrow r) \in \text{mn}(t)$ .

*Example 16.* Continuing Example 12, we have  $\text{mn}(t') = \{2\}$  and, thus, the natural narrowing sequence  $\text{B}(\mathbf{X}, \underline{\text{B}(\mathbf{F}, \mathbf{T}, \mathbf{T})}, \mathbf{F}) \xrightarrow[\text{id}]{m} \underline{\text{B}(\mathbf{X}, \mathbf{T}, \mathbf{F})} \xrightarrow[\text{id}]{m} \mathbf{T}$

Similarly to the rewriting case, we prove that both definitions of natural narrowing coincide for safe *mdt*'s.

**Theorem 2.** *Let  $\mathcal{R} = (\mathcal{F}, R)$  be a left-linear CS,  $f \in \mathcal{D}$ ,  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  s.t.  $\text{root}(t) = f$ , and  $T_f$  be a safe *mdt* for  $f$ . Then,  $\text{mnt}(t, T_f) = \text{mn}(t)$ .*

## 5 Implementation

In this section, we show that natural rewriting and natural narrowing can be implemented efficiently by reusing modern state-of-the-art compilation techniques developed for (weakly) outermost-needed rewriting and narrowing. Curry [12] is a multiparadigm programming language based on weakly outermost-needed narrowing which combines purely functional programming, purely logic programming, and concurrent (logic) programming in a seamless way. The Curry2Prolog compiler [6] included into the Curry development system PAKCS [11] is one of the fastest Curry implementations available nowadays. The following example helps to understand how the compilation of Curry programs to Prolog [6] works.

*Example 17.* Consider the TRS of Example 2. Basically, the technique in [6] translates a function symbol and its associated definitional tree to **case** expressions and, then, compiles these **case** expressions to Prolog. For instance, Curry2Prolog uses the definitional tree (a) of Figure 1 and translates it to the following (auxiliary) case expressions, where the constructor symbol 0 is represented by the constructor symbol **z**:

$$\begin{aligned} \text{M} \div \text{N} = \text{case M of z} &\quad \rightarrow (\text{case N of s(N')} \rightarrow \text{z}) \\ &\quad \text{s(M')} \rightarrow (\text{case N of s(N')} \rightarrow \text{s(M' - N'} \div \text{s(N')})) \end{aligned}$$

Its translation to Prolog according to [6] is as follows:

```
:- block ÷(?,?,?,-,?).
÷(M,N,Result,Ein,Eout):-hnf(M,HM,Ein,E1),÷_1(HM,N,Result,E1,EOut).

:- block ÷_1(?,?,?,-,?).
÷_1(z,N,Result,Ein,Eout):-hnf(N,HN,Ein,E1),÷_1_z_2(HN,Result,E1,Eout).
÷_1(s(M),N,Res,Ein,Eout):-hnf(N,HN,Ein,E1),÷_1_s_2(HN,M,Res,E1,Eout).

:- block ÷_1_z_2(?,?,-,?).
÷_1_z_2(s(N),z,E,E).

:- block ÷_1_s_2(?,?,?,-,?).
÷_1_s_2(s(N),M,s(÷(- (M,share(N,EN,RN)),s(share(N,EN,RN))))),E,E).
```

As is usual in the transformation of functions to predicates, the result of the function call is included as an extra argument called **Result**. Since the computational model of Curry allows concurrent executions of calls, the Curry2Prolog

compiler introduces `block` Prolog declarations and arguments `Ein` and `Eout` for each function symbol in order to control whether a function call is suspended or not (see [6]). Moreover, a call `hnf(T,HT,Ein,Eout)` is responsible for obtaining the head normal form `HT` of a term `T`. Finally, `Curry2Prolog` implements the sharing of variables using an extra symbol `share`. A term `share(T,ET,RT)` contains the shared term `T`, `ET` (which indicates whether `T` has already been evaluated), and `RT` (which is the result of `T`).

In this section, we argue that it is possible to use the technique in [6] to translate matching definitional trees and natural rewriting and narrowing strategies to Prolog. The main difference between a *gdt* and a *mdt* is that the latter has more than one inductive position. Hence, we can use the previous technique for translating each inductive position and its set of subtrees to Prolog and include some extra rules when more than one inductive position exist. These extra rules check sequentially whether each inductive position is constructor-rooted or not and only start an evaluation if none of the inductive positions are constructor-rooted (according to Definition 4). These rules use a predicate `checkC` that succeeds when its argument is rooted by a constructor symbol. We show how the translation works in the following example.

*Example 18.* Consider the TRS in Example 2 and the safe *mdt* in Example 5 (and Figure 3). We can apply the technique in Example 17 to each part of the root *branch*-node driven by an inductive position, and encode the pattern matching process into an appropriate set of rules (using predicate `hnf`). Then we add the necessary rules (using predicate `checkC`) to provide the behavior associated to the natural narrowing strategy. Note that no extra rules (using `checkC`) are necessary when there exists only one inductive position in a *branch*-node. The compilation to Prolog yields:

```

checkC(A):-var(A),!,fail.
checkC(z).
checkC(s(_)).

:- block ÷(?,?,?,-,?).
÷(M,N,Result,Ein,Eout):-checkC(M),!,÷_1(M,N,Result,Ein,Eout).
÷(M,N,Result,Ein,Eout):-checkC(N),!,÷_2(N,M,Result,Ein,Eout).
÷(M,N,Result,Ein,Eout):-hnf(M,HN,Ein,E1),÷_1(HN,N,Result,E1,Eout).

:- block ÷_1(?,?,?,-,?).
÷_1(z,N,Result,Ein,Eout):-hnf(N,HN,Ein,E1),÷_1_z_2(HN,Result,E1,Eout).
÷_1(s(M),N,Res,Ein,Eout):-hnf(N,HN,Ein,E1),÷_1_s_2(HN,M,Res,E1,Eout).

:- block ÷_1_z_2(?,?,-,?).
÷_1_z_2(s(N),z,E,E).

:- block ÷_1_s_2(?,?,-,?).
÷_1_s_2(s(N),M,s(÷(-M,share(N,EN,RN)),s(share(N,EN,RN))))),E,E).

:- block ÷_2(?,?,?,-,?).
÷_2(s(N),M,Res,Ein,Eout):-hnf(M,HN,Ein,E1),÷_2_s_1(HN,N,Res,E1,Eout).

```

**Table 1.** Runtimes (in ms.) of different calls within Prolog compiled programs

Benchmark	Goal	PAKCS	PAKCS with Natural Narrowing
$\div$	$10! \div 10$	26632	27345
	$\perp \div 0$	$\infty$	0
$\leq$	$10! \leq 10!$	31360	31212

```

:- block  $\div\_2\_s\_1(?,?,-,?)$ .
 $\div\_2\_s\_1(z,N,z,E,E)$ .
 $\div\_2\_s\_1(s(M),N,s(\div(-M,share(N,EN,RN)),s(share(N,EN,RN))))$ ,E,E).

```

Here, the predicate `checkC` is used to perform the constructor-rooted test on all the inductive positions. Note that we do not allow backtracking (using the Prolog cut `!`) in the selection of an inductive position and a subtree. Note also that a simple optimization is directly performed in the previous example: “when two inductive positions are demanded by the same set of rules, it is sufficient to evaluate only one of them.” Thus, only position 1 is evaluated when both 1 and 2 are demanded (see the third clause of predicate  `$\div$` ).

Finally, Table 1 shows that the compiled Prolog code of the natural narrowing strategy using matching definitional trees does not introduce any appreciable overhead while providing better computational properties. It shows the average in milliseconds of 10 executions measured on a Pentium III machine running RedHat 7.2. The Prolog code for symbol  `$\div$`  in the third column corresponds to the Prolog code in Example 17, whereas the Prolog code for the fourth column corresponds to the Prolog code in Example 18. However, the Prolog code for symbol  `$\leq$`  is the same for the third and fourth columns since its *mdt* is translated to the same Prolog code as the code produced by [6]. Thus, the execution times for symbol  `$\leq$`  are similar. Note that the non-terminating symbol  `$\perp$`  is defined by the rule  `$\perp = \perp$` , and the mark  `$\infty$`  represents an infinite evaluation sequence.

In future work, we plan to perform more exhaustive experiments.

## 6 Conclusions

We have provided an extension of the notion of definitional tree, called *matching definitional tree*, and a reformulation of natural rewriting and narrowing using these matching definitional trees. We have proved that both reformulations are correct w.r.t. to their original definitions. Moreover, we have shown that it is possible to implement natural rewriting and narrowing efficiently by compiling to Prolog. Hence, it seems possible to include the natural rewriting and narrowing strategies into current existing implementations without great effort. This could encourage programmers to write non-inductively sequential programs, whose sequential parts could still be executed in an optimal way.

*Acknowledgements.* I am grateful to María Alpuente, Salvador Lucas, and the anonymous referees for their helpful remarks.

## References

1. M. Alpuente, S. Escobar, B. Gramlich, and S. Lucas. Improving on-demand strategy annotations. In M. Baaz and A. Voronkov, editors, *Proc. of the 9th Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'02)*, Springer LNCS 2514, pages 1–18, 2002.
2. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of lazy functional logic programs. In *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97, ACM Sigplan Notices*, 32(12):151–162, ACM Press, New York, 1997.
3. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conf. on Algebraic and Logic Programming ALP'92*, Springer LNCS 632, pages 143–157, 1992.
4. S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. of the Fourteenth Int'l Conf. on Logic Programming (ICLP'97)*, pages 138–152. MIT Press, 1997.
5. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Journal of the ACM*, 47(4):776–822, 2000.
6. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into prolog. In *Proc. of the 3rd Int'l Workshop on Frontiers of Combining Systems (FroCoS 2000)*, Springer LNCS 1794, pages 171–185, 2000.
7. S. Antoy and S. Lucas. Demandness in rewriting and narrowing. In M. Comini and M. Falaschi, editors, *Proc. of the 11th Int'l Workshop on Functional and (Constraint) Logic Programming WFLP'02*, Elsevier ENTCS 76, 2002.
8. T. Arts and J. Giesl. A collection of examples for termination of term rewriting using dependency pairs. Technical report, AIB-2001-09, RWTH Aachen, Germany, 2001.
9. S. Escobar. Refining weakly outermost-needed rewriting and narrowing. In D. Miller, editor, *Proc. of the 5th Int'l ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, PPDP'03*, pages 113–123. ACM Press, New York, 2003.
10. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
11. M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS 1.5.0: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany, 2003.
12. M. Hanus, S. Antoy, H. Kuchen, F. López-Fraguas, W. Lux, J. Moreno Navarro, and F. Steiner. Curry: An Integrated Functional Logic Language (version 0.8). Available at: <http://www.informatik.uni-kiel.de/~curry>, 2003.
13. G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems, Part I + II. In *Computational logic: Essays in honour of J. Alan Robinson*, pages 395–414 and 415–443. The MIT Press, Cambridge, MA, 1992.
14. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.
15. R. Sekar and I. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. *Information and Computation*, 104(1):78–109, 1993.
16. TeReSe, editor. *Term Rewriting Systems*. Cambridge University Press, Cambridge, 2003.