In *Proc. of the Fourteenth International Conference on Logic Programming*, Leuven (Belgium), July 1997, MIT Press, pp. 138-152

Parallel Evaluation Strategies for Functional Logic Languages

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

Rachid Echahed
IMAG-LSR, CNRS, BP 53, F-38041 Grenoble, France
echahed@imag.fr

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

Abstract

We introduce novel, sound, complete, and locally optimal evaluation strategies for functional logic programming languages. Our strategies combine, in a non-trivial way, two landmark techniques in this area: the computation of unifiers performed by needed narrowing in inductively sequential rewrite systems and the simultaneous reduction of a necessary set of redexes performed by rewriting in weakly orthogonal, constructor-based rewrite systems. First, we define a sequential strategy similar in scope to other narrowing strategies used in modern lazy functional logic languages. Then, based on the sequential strategy, we define a parallel narrowing strategy that has several noteworthy characteristics: it is the first complete narrowing strategy which evaluates ground expressions in a fully deterministic, optimal way; it computes shortest derivations and minimal sets of solutions on inductively sequential rewrite systems; and when combined with term simplification, it subsumes and improves all recently developed optimizations of narrowing for overlapping rewrite rules.

1 Introduction

The interest in integrating functional and logic programming has grown over the last decade, since the languages resulting from this integration are expected to have advantages of both paradigms. Most proposals with a sound and complete operational semantics for the integration of functional and logic programming languages (see [10] for a recent survey) are based on *narrowing*. Narrowing *solves* equations by computing unifiers with respect to an equational theory. 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 addition for natural numbers, which are represented by terms built with 0 and s:

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

To narrow the equation $Z+s(0) \approx s(s(0))$, rule R_2 is applied by instantiating Z to $s(X)$. To narrow the resulting equation, $s(X+s(0)) \approx s(s(0))$, R_1 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 programs. Thus, many narrowing strategies for limiting the size of the search space have been proposed [10]. Recently, an optimal strategy for inductively sequential rewrite systems (e.g., the rewrite system in Example 1) has been discovered by extending to narrowing landmark results in term rewriting [2]. In this paper, we investigate new evaluation strategies for a more general class of programs, namely those defined by weakly orthogonal, constructor-based systems.

Example 2 Consider the following definition of Boolean disjunction known as *parallel-or*.

$$\begin{array}{rcll} X \vee \mathit{true} & \to & \mathit{true} & \mathrm{R}_1 \\ \mathit{true} \vee X & \to & \mathit{true} & \mathrm{R}_2 \\ \mathit{false} \vee \mathit{false} & \to & \mathit{false} & \mathrm{R}_3 \end{array} \tag{1}$$

A significant difference of this system w.r.t. the previous one is the overlapping of the first two rules. As a consequence, a term of the form $t_1 \vee t_2$ may be narrowed to normal form by narrowing either t_1 or t_2, although we do not know of any criterion to make this choice without look-ahead.

To place our results in a context, we briefly review relevant results about rewriting strategies. O'Donnell has shown [19] that the parallel outermost strategy is normalizing for almost orthogonal TRSs, hence for weakly orthogonal, constructor-based TRSs. In general, some reductions performed by this strategy could be avoided. Huet and Lévy have shown [11] that for the class of strongly sequential TRSs there is an effective strategy that performs only unavoidable reductions. Sekar and Ramakrishnan [21] have refined O'Donnell's result in a different direction. Within the class of the weakly orthogonal, constructor-based TRSs, they have shown that it is possible to minimize the set of redexes that must be reduced in parallel in a term to compute its normal form. The resulting strategy, similar to Huet and Lévy's, does not take into account the right hand sides of the TRS's rules, and it is optimal among the strategies with this limitation.

To date, only one narrowing strategy generalizes a rewriting strategy. Huet and Lévy's approach has been extended to narrowing for inductively-sequential TRSs with comparable properties. The resulting strategy, called *needed* [2], performs only unavoidable steps and turns out to be optimal also with respect to the computed unifiers. However, narrowing strategies for weakly orthogonal TRSs depart radically from O'Donnell's and Sekar and Ramakrishnan's approaches in that they are sequential. This departure has a major impact on the operational meaning of completeness of a strategy.

If a ground term t has a normal form, then both O'Donnell's and Sekar and Ramakrishnan's strategies compute the normal form of t by means of deterministic, parallel[1] steps. Narrowing t is equivalent to rewriting it, since

[1] In this context, *parallel* means that several, possibly different redexes are simultaneously reduced in a single step.

we are assuming that t is ground. All the existing narrowing strategies that are known to be ground complete narrow t to its normal form by means of possibly *don't-know* non-deterministic, sequential steps. This notion of completeness is somewhat reductive in the sense that the implementations of these strategies *don't know* how to compute the normal form of t without a severe penalty in efficiency. However, this need not be the case for all ground and for some non-ground terms.

The subject of this paper is a parallel strategy for narrowing. Our strategy is sound and complete and can be implemented relatively efficiently by unification. It *always* computes the normal form of a ground term, if there exists one, without non-determinism and as efficiently as possible under a set of reasonable assumptions. Our strategy narrows a necessary set of positions, which generally contains fewer than all the outermost narrowable positions of a term. Our parallel strategy falls back to the *needed narrowing strategy* [2] on the inductively sequential portions of a TRS, and consequently is optimal on these portions, and falls back to Sekar and Ramakrishnan's strategy on the ground terms, and consequently is optimal (in a weaker sense) on the ground portions of a computation, too. Note that our parallel narrowing strategy is not intended as a technique to implement functional logic languages on parallel architectures, since the parallelism is too fine-grained. The parallelism is mainly used to avoid some redundant non-deterministic choices of simpler narrowing strategies.

The paper is organized as follows. Some preliminary definitions and notations are listed in the next section. Section 3 defines the *weakly needed rewriting strategy* which is a parallel rewriting strategy designed for the class of weakly orthogonal, constructor-based TRSs. In Section 4, we present a *sequential* narrowing strategy which is a natural extension of needed narrowing to overlapping TRSs. We define the *parallel narrowing strategy* and an important improvement in Sections 5 and 6 and discuss its optimality in Section 7. Comparison with related work is given in Section 8. Due to lack of space, some detailed definitions and all proofs are omitted from this paper. A full version containing all the details can be found in [3].

2 Preliminaries

We recall some key notions and notations about rewriting. We are consistent with the conventions of [5, 13].

Terms are constructed w.r.t. a given many-sorted *signature* Σ. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term t is called *ground* if $\mathcal{V}ar(t) = \varnothing$. In practice, most functional logic programs are *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 functional logic languages *ALF* [8], *BABEL* [18], *K-LEAF* [7], *LPG* [4]). Hence, we assume that $\mathcal{R}$ is a *constructor-based term rewriting system* consisting of *rewrite rules* $l \to r$, where l is a *pattern*, i.e., the root of l is an operation symbol and the arguments of l do not contain any operation symbol. A term $f(t_1, \ldots, t_n)$ $(n \geq 0)$ is called an *operation-rooted term* if f is an operation.

Substitutions and unifiers are defined as usual [5], where we write $mgu(s, t)$ for the *most general unifier* of s and t. We write $\sigma \equiv_V \theta$ iff the substitutions

σ and θ are renamed variants on the set V. We write $t \leq t'$ (respectively, $\sigma \leq_V \sigma'$) iff there is a substitution τ such that $t' = \tau(t)$ (respectively, $\sigma'(x) = \tau(\sigma(x))$ for all variables $x \in V$).

An *occurrence* or *position* p is a sequence of positive integers identifying a subterm in a term. $t_{|p}$ denotes the subterm of t at position p, and the result of *replacing* $t_{|p}$ with s in t is denoted by $t[s]_p$. We write $p \leq q$ to denote that the position p is a *prefix* of q.

A *reduction step* is an application of a rewrite rule $l \to r$ to the redex $t_{|p}$, i.e., $t \to_{p,l\to r} s$ if $s = t[\sigma(r)]_p$ for some substitution σ with $t_{|p} = \sigma(l)$. $\overset{*}{\to}$ denotes the transitive and reflexive closure of $\to$. A term t is *reducible to* a term s if $t \overset{*}{\to} s$. A term t is called *irreducible* or in *normal form* if there is no term s with $t \to s$. A term rewriting system $\mathcal{R}$ is called *terminating* if there are no infinite rewrite derivations w.r.t. $\mathcal{R}$.

Rewriting is computing the *value* of a functional expression, i.e., its normal form obtained by rewriting. Functional logic programs compute with partial information, i.e., a functional expression may contain logic variables. The goal is to compute values for these variables such that the expression is evaluable to a particular normal form, e.g., a constructor term [4, 7, 18]. This is done by narrowing. 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 $\mathcal{V}ar(t) \cap \mathcal{V}ar(l \to r) = \varnothing$ and a unifier σ of $t_{|p}$ and l such that $s = \sigma(t[r]_p)$. In this case we write $t \leadsto_{p,l\to r,\sigma} s$, where p and $l \to r$ are sometimes omitted. If σ is a most general unifier of $t_{|p}$ and l, the narrowing step is called *most general*. 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 most papers, narrowing is intended as most general narrowing [10]. Most general narrowing has the advantage that most general unifiers are uniquely computable, whereas there exist many independent unifiers. However, as shown in [2], most general unifiers must be dropped to obtain an optimal narrowing strategy. This paper follows the same approach.

Narrowing is intended to solve goals, where a *goal* is a Boolean expression that should be reduced to the constant *true*. Thus, a substitution σ is a *solution* for a goal G iff $\sigma(G)$ is reducible to *true*. This is general enough to cover the equation solving capabilities of current functional logic languages with a lazy operational semantics, like BABEL [18] or K-LEAF [7], since the strict equality[2] $\approx$ can be defined as a binary operation by a set of orthogonal rewrite rules (see [2, 7, 18] for more details about strict equality). An important consequence of restricting narrowing to goals is the fact that during the successful rewriting of a goal the topmost symbol is always an operation or the constant *true*. This property will be used to simplify the presentation of our results. Note that the evaluation of an arbitrary term t to a constructor normal form can be obtained by solving the goal $t \approx X$.

To ensure the confluence of the rewrite relation, we also require weak orthogonality. A term rewriting system $\mathcal{R}$ is *weakly orthogonal* if for each rule $l \to r \in \mathcal{R}$ the left-hand side l does not contain multiple occurrences of a variable (*left-linearity*) and for each pair of rules $l \to r, l' \to r' \in \mathcal{R}$,

[2]The *strict equality* $t \approx t'$ holds if t and t' are reducible to the same ground constructor term. Note that normal forms may not exist in general due to non-terminating rewrite rules.

non-variable subterm $l_{|p}$ of l, and mgu σ for $l_{|p}$ and l', the terms $\sigma(l[r']_p)$ and $\sigma(r)$ are identical. $\mathcal{R}$ is *almost orthogonal* if it is weakly orthogonal and for each pair of rules $l \to r, l' \to r' \in \mathcal{R}$, the only possible non-variable subterm of l that may unify with l' is l itself. Since we consider in the following only **C**onstructor-based, **A**lmost orthogonal, **T**erm rewriting systems, we write *CAT* for this class.

It is easy to see that for constructor-based systems almost and weak orthogonality are the same concept, since the left-hand sides of the rules are patterns. The notion of *descendant*, well-known for orthogonal systems [11], is extended to almost orthogonal systems without difficulties. Here we provide an intuitive definition as proposed in [14]. 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. A position u of a term t is called *needed* iff in every reduction sequence of t to a normal form a descendant of $t_{|u}$ is rewritten at its root.

Example 3 Consider the rewrite rule $\mathrm{R}_3 = double(X) \to X+X$. The following reduction of $double(0+0)$ shows, by means of underlining, the descendants of $0+0$.

$$double(0 \underline{\ +\ } 0) \to_{\Lambda, \mathrm{R}_3} (0 \underline{\ +\ } 0) + (0 \underline{\ +\ } 0)$$

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

3 Weakly Needed Rewriting

For inductively sequential systems there exists a narrowing strategy [2] that performs only steps that are needed for solving goals. This strategy is a generalization to narrowing of the sequential rewriting strategy presented in [1]. This sequential strategy is also the basis of a parallel rewriting strategy for weakly orthogonal, constructor-based rewrite systems, referred to as *weakly needed rewriting* and sketched first in [1], that computes the same reduction sequences of [21], although the overall approach is different. In this section, we reformulate the weakly needed rewriting strategy and show one important property of this generalization. We begin with some technical definitions.

A definitional tree is a hierarchical structure containing the rules of a defined operation of a rewrite system. The symbols *rule*, *branch*, and *or* occurring in the next definition, are uninterpreted functions used to classify the nodes of the tree. A definitional tree can be seen as a partially ordered set of patterns with some additional constraints.

Definition 1 $\mathcal{T}$ is a *generalized definitional tree*, or *gdt*, with pattern π iff the depth of $\mathcal{T}$ is finite, π is a pattern, and one of the following cases holds:

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

$\mathcal{T} =$ $branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, where o is an occurrence of a variable in π, $c_1, \ldots, c_k$ are different constructors of the sort of $\pi_{|o}$, for some $k > 0$, and, for all j in $\{1, \ldots, k\}$, $\mathcal{T}_j$ is a *gdt* with pattern $\pi[c_j(X_1, \ldots, X_n)]_o$, where n is the arity of c_j and $X_1, \ldots, X_n$ are new variables.

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

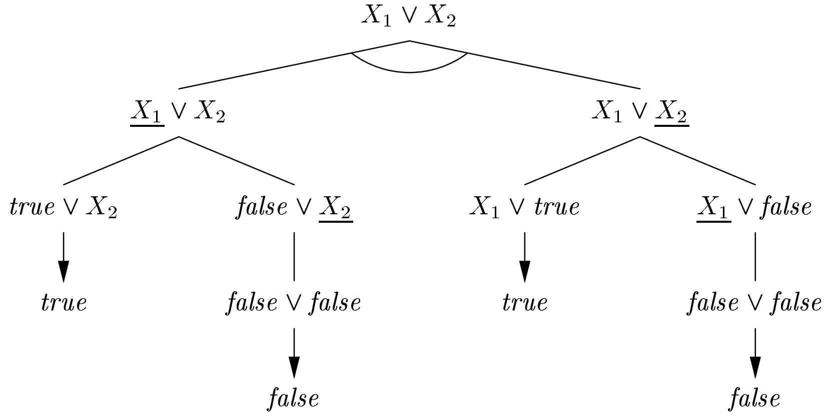

Figure 1: Pictorial representation of a parallel definitional tree of the operation *parallel-or* defined in display (1). The branch variables in the patterns of *branch* nodes are underlined. *Or*-ed branches are joined by an arc.

In the remainder of the paper, we will use the notation $pattern(\mathcal{T})$ to denote the pattern argument of a *gdt* $\mathcal{T}$.

Let $\mathcal{R}$ be a rewrite system. $\mathcal{T}$ is a *gdt of an operation* f iff $\mathcal{T}$ is a *gdt* such that $pattern(\mathcal{T}) = f(X_1, \ldots, X_n)$, where n is the arity of f and $X_1, \ldots, X_n$ are new distinct variables, and for every rule $l \to r$ of $\mathcal{R}$ with $l = f(t_1, \ldots, t_n)$ there exists a leaf $rule(l' \to r')$ of $\mathcal{T}$ such that l is a variant of l'.

A generalized definitional tree $\mathcal{T}$ is called *parallel definitional tree*, abbreviated *pdt*, iff in every node $or(\mathcal{T}_1, \ldots, \mathcal{T}_k)$ every $\mathcal{T}_j$ has a *branch* node at the top, where these *branch* nodes contain pairwise different positions.

A *definitional tree* is a generalized definitional tree without *or*-nodes.[3]

Figure 1 pictorially represents the parallel definitional tree of the rules of the parallel-or shown in Example 2. It is easy to see that a generalized definitional tree exists for each operation. A parallel definitional tree may not exist if the rewrite system contains *useless rules*, i.e., rules that are instances of another rule. By eliminating all the useless rules from a rewrite system $\mathcal{R}$, every operation of the resulting system has a parallel definitional tree which can be effectively constructed [1, Th. 19]. Moreover, the rewrite relation and the set of solutions is not changed by this elimination. From now on, we assume that every rewrite system that we are dealing with has no useless rules.

A parallel definitional tree may be decomposed into a set of *sequential components* each of which is a (sequential) definitional tree. If $\mathcal{T} = rule(l \to r)$, then $\mathcal{T}$ itself is the only *sequential component* of $\mathcal{T}$. If $\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, then $branch(\pi, o, \mathcal{T}'_1, \ldots, \mathcal{T}'_k)$ is a *sequential component* of $\mathcal{T}$ for all sequential components $\mathcal{T}'_j$ of $\mathcal{T}_j$, $j = 1, \ldots, k$. If $\mathcal{T} = or(\mathcal{T}_1, \ldots, \mathcal{T}_k)$, then, for all sequential components $\mathcal{T}'$ of $\mathcal{T}_j$, $\mathcal{T}'$ is a *sequential component* of $\mathcal{T}$.

Below, we recall the definition of needed rewriting. Needed rewriting is a strategy for *inductively sequential systems*, i.e., rewrite systems where each function has a definitional tree. Loosely speaking, the rewriting (and narrowing) strategies presented in this paper are obtained by breaking up

[3] This corresponds to the definition given in [2] except that we ignore the *exempt* nodes.

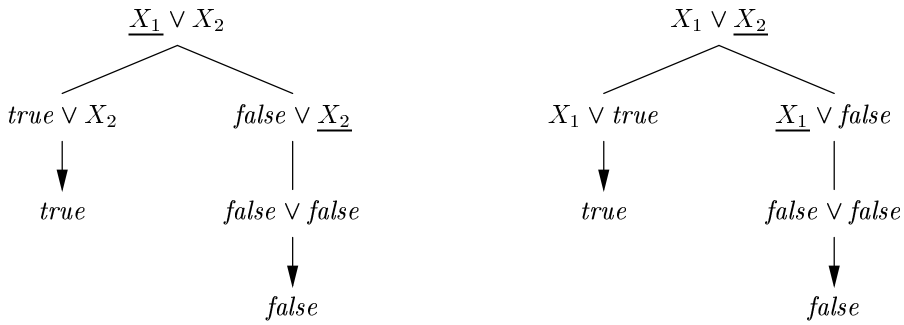

Figure 2: Pictorial representation of the sequential components of the parallel definitional tree of the operation *parallel-or* defined in display (1). Each component is a sequential definitional tree and is obtained by taking one distinct subtree of the *or* node at the root in Fig. 1.

a CAT into its inductively sequential components, applying needed rewriting (or narrowing) to each component, and combining together the results of each application.

The needed rewriting strategy is implemented by a function, φ, that takes two arguments, an operation-rooted term, t, and a definitional tree, $\mathcal{T}$, of the root of t. Throughout an interleaved descent down both t and $\mathcal{T}$, φ computes, whenever possible, a position p and a rule R such that t must be reduced at p, using rule R, to compute its constructor normal form.

Definition 2 The partial function φ takes two arguments, an operation-rooted term t and a definitional tree $\mathcal{T}$ such that $pattern(\mathcal{T}) \leq t$. If $\varphi(t, \mathcal{T})$ is defined, it yields a pair, (p, R), where p is a position of t and R is a rewrite rule applicable to t at p. The function φ is defined recursively as follows

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

In order to extend the strategy φ to CATs, we apply φ to all the sequential components of a *pdt* and select the disjoint outermost positions from all computed positions. This strategy is denoted by $\bar{\varphi}$. The *weakly needed rewriting* strategy reduces all redexes at positions computed by $\bar{\varphi}$ in parallel.

Example 4 Consider the rewrite system of Example 2 and the term $t = (true \vee (true \vee true)) \vee (X \vee (false \vee false))$. The weakly needed rewrite derivation computed by $\bar{\varphi}$ is

$$t \to_{(1,R_2),(2\cdot 2,R_3)} true \vee (X \vee false) \to_{(\Lambda,R_2)} true$$

The following theorem shows that first, unless we perform at least one reduction step computed by $\bar{\varphi}$ we cannot obtain the normal form and second, that if we perform all the steps computed by $\bar{\varphi}$ we do obtain the normal form (whenever it is a constructor term) of a goal.

Theorem 1 *Let $\mathcal{R}$ be a CAT, and G a goal which is reducible to 'true'.*
1. *Every strategy normalizing G must reduce a descendant of G at some position computed by $\bar{\varphi}$.*
2. *A strategy $\mathcal{S}$ that reduces the descendants of the redexes computed in G by $\bar{\varphi}$ is normalizing.*

Thus, $\bar{\varphi}$ computes a necessary set of redexes in the sense of [21], although the way in which the set is computed, i.e. by means of φ, is quite different. We define in the next section a generalization of φ, λ, that simultaneously computes both a redex and a unifier. This allows us to generalize to narrowing the results of [21] on rewriting.

4 Weakly Needed Narrowing

In this section we study our first narrowing strategy for CATs. This strategy is sequential and could be seen as a natural extension to overlapping TRSs of needed narrowing [2]. In order to define the narrowing steps, we use the sequential components of a parallel definitional tree. Loosely speaking, we apply the needed narrowing strategy λ (defined in [2] and recalled below) to all the sequential components of a *pdt* and combine the results together. Since λ computes optimal narrowing derivations for inductively sequential programs, our strategy is a conservative extension of an optimal strategy.

Definition 3 The function λ takes two arguments, an operation-rooted term t and a definitional tree $\mathcal{T}$ 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 a rewrite rule, $l \to r$, of $\mathcal{R}$ and σ is a unifier of l and $t_{|p}$. Thus, let t be a term and $\mathcal{T}$ a definitional tree in the domain of λ. The function λ is defined to yield least sets of triples satisfying the following conditions.

$$\lambda(t, \mathcal{T}) \supseteq \begin{cases} \{(\Lambda, l \to r, mgu(t, l))\} & \text{if } \mathcal{T} = rule(l \to r); \\ \lambda(t, \mathcal{T}_i) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad t \text{ and } pattern(\mathcal{T}_i) \text{ unify, for some } i; \\ \{(o \cdot p, R, \sigma \circ \tau)\} & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad t_{|o} \text{ is operation-rooted, } \tau = mgu(t, \pi), \\ & \quad \mathcal{T}' \text{ is a definitional tree of the root of} \\ & \quad \tau(t_{|o}), \text{ and } (p, R, \sigma) \in \lambda(\tau(t_{|o}), \mathcal{T}'). \end{cases}$$

If $(p, l \to r, \sigma) \in \lambda(t, \mathcal{T})$, then $t \rightsquigarrow_{p, l \to r, \sigma} \sigma(t[r]_p)$ is a narrowing step. As in proof procedures for logic programming, we have to apply *variants* of the rewrite rules *with fresh variables* at each narrowing step, i.e., the definitional trees always contain new variables if they are used in a narrowing step.

Definition 4 The function $\bar{\lambda}$ takes two arguments, an operation-rooted term t and a *pdt* $\mathcal{T}$ such that $pattern(\mathcal{T})$ and t unify. Then, $\bar{\lambda}$ is defined by

$$\bar{\lambda}(t, \mathcal{T}) = \{(p, R, \sigma) \in \lambda(t, \mathcal{T}') \mid \mathcal{T}' \text{ is a sequential component of } \mathcal{T}\}$$

We call *weakly needed* any narrowing step $t \rightsquigarrow_{p, R, \sigma} t'$ with $(p, R, \sigma) \in \bar{\lambda}(t, \mathcal{T})$.

Weakly needed narrowing is almost identical to the demand driven narrowing strategy proposed in [16]. However, soundness and completeness results are not provided in [16].

Example 5 Consider Example 2 with the additional rule $\mathrm{R}_4 = f(a) \to \mathit{true}$ and the term $t = f(X) \vee f(X)$. Let $\mathcal{T}$ denote the parallel definitional tree of "$\vee$" pictorially represented in Fig. 1. The sequential components of $\mathcal{T}$ are pictorially represented in Fig. 2. According to Definition 4, $\bar{\lambda}(t, \mathcal{T})$ is

$$\{(1, \mathrm{R}_4, \{X \mapsto a\}), (2, \mathrm{R}_4, \{X \mapsto a\})\}$$

which specifies the following narrowing steps:

$$t \rightsquigarrow_{1, \mathrm{R}_4, \{X \mapsto a\}} \mathit{true} \vee f(a)$$
$$t \rightsquigarrow_{2, \mathrm{R}_4, \{X \mapsto a\}} f(a) \vee \mathit{true}$$

Theorem 2 (Soundness of weakly needed narrowing) *Let $\mathcal{R}$ be a CAT and G a goal. If $G \rightsquigarrow_{\sigma_1} \cdots \rightsquigarrow_{\sigma_n} \mathit{true}$ is a narrowing derivation computed by $\bar{\lambda}$, then $\sigma_n \circ \cdots \circ \sigma_1$ is a solution for G.*

The completeness of weakly needed narrowing is stated w.r.t. *constructor substitutions* as solutions of goals, i.e., substitutions mapping variables into constructor terms. 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 [12], and lazy narrowing is complete only for constructor substitutions [7, 18].

Theorem 3 (Completeness of weakly needed narrowing) *Let $\mathcal{R}$ be a CAT. Let σ be a constructor substitution that is a solution of a goal G and V be a finite set of variables containing $\mathcal{V}ar(G)$. Then $\bar{\lambda}$ computes a narrowing derivation $G \rightsquigarrow_{\sigma_1} \cdots \rightsquigarrow_{\sigma_n} \mathit{true}$ such that $\sigma_n \circ \cdots \circ \sigma_1 \leq_V \sigma$.*

If we consider again the term t in Example 5, we can observe that, to narrow t to *true*, the strategy $\bar{\lambda}$ computes four distinct derivations with the same substitution $\{X \mapsto a\}$. In order to avoid such redundant computations, we propose a parallel narrowing strategy in the next section.

5 Parallel Narrowing

Classic narrowing may be defined in two steps as follows: t narrows to t' iff there exists a substitution σ such that the term $\sigma(t)$ rewrites to t' using some rewrite rule $l \to r$. From this informal definition, narrowing differs from rewriting only by the instantiation step. Now, if we generalize this idea to parallel rewriting, i.e., if we replace the rewriting step, in the narrowing relation, by a parallel rewriting step, we obtain a new relation that we call *parallel narrowing*. The definition below formalizes the idea that we just sketched and defines a parallel narrowing step as an instantiation followed by a parallel rewriting step.

Definition 5 Let $\mathcal{R}$ be a term rewriting system and $\mathcal{S}$ a parallel rewriting strategy. $t \overset{\mathcal{S}}{\rightsquigarrow}_{\sigma} t'$ is a *parallel narrowing* step (w.r.t. $\mathcal{S}$) iff $\sigma(t) \overset{\mathcal{S}}{\to} t'$. A *parallel narrowing strategy* $\mathcal{N}_\mathcal{S}$ is a function from terms to sets of substitutions, $\mathcal{N}_\mathcal{S} : \mathcal{T}(\Sigma, \mathcal{X}) \to 2^{Sub}$. A substitution σ is in $\mathcal{N}_\mathcal{S}(t)$ only if there exists a term t' such that $t \overset{\mathcal{S}}{\rightsquigarrow}_{\sigma} t'$. We denote the parallel narrowing relation w.r.t. a strategy $\mathcal{N}_\mathcal{S}$ by $\overset{\mathcal{N}_\mathcal{S}}{\rightsquigarrow}$.

Throughout this section, parallel narrowing is defined upon the parallel rewriting strategy $\bar{\varphi}$. Below we define the parallel narrowing strategy $\bar{\bar{\lambda}}$. There are two main differences w.r.t. weakly needed narrowing: $\bar{\bar{\lambda}}$ may disregard some unifiers computed by weakly needed narrowing which contribute to redundant derivations, and at every narrowing step a necessary set of redexes of the instantiated term is reduced in parallel.

Definition 6 Let $\mathcal{R}$ be a CAT, t an operation-rooted term, $\mathcal{T}$ a parallel definitional tree of the root of t. We define the parallel narrowing strategy $\bar{\bar{\lambda}}$ as follows:[4]

$$\begin{aligned}\bar{\bar{\lambda}}(t,\mathcal{T}) = \{\sigma_{|\mathcal{V}ar(t)} \mid \exists\, (p,R,\sigma) \in \bar{\lambda}(t,\mathcal{T}),\ \forall\, (q,R',\theta) \in \bar{\lambda}(t,\mathcal{T}),\\ (\theta \leq_{\mathcal{V}ar(t)} \sigma \text{ and } \theta \not\equiv_{\mathcal{V}ar(t)} id \Rightarrow \sigma \equiv_{\mathcal{V}ar(t)} \theta) \text{ and}\\ (\theta \equiv_{\mathcal{V}ar(t)} id \text{ and } q \leq p \Rightarrow \sigma \equiv_{\mathcal{V}ar(t)} id)\} \,/\, \equiv\end{aligned}$$

Intuitively, a substitution σ belongs to $\bar{\bar{\lambda}}(t,\mathcal{T})$ iff σ is either the identity or a minimal substitution (w.r.t. $\leq$) among the non-identity substitutions computed by $\bar{\lambda}(t,\mathcal{T})$. Furthermore, whenever two triples (p,R,id) and (q,R',θ) belong to $\bar{\lambda}(t,\mathcal{T})$ with p being a prefix of q ($p \leq q$), the substitution θ is not considered by the strategy $\bar{\bar{\lambda}}$.

Example 6 Consider the following rewrite rules:

$$\begin{array}{rcll} X * 0 & \to & 0 & \mathrm{R}_1 \\ 0 * X & \to & 0 & \mathrm{R}_2 \end{array} \qquad \begin{array}{rcll} f(s(s(X))) & \to & 0 & \mathrm{R}_3 \\ g(X) & \to & g(s(X)) & \mathrm{R}_4 \end{array}$$

and the term $t = g(X) * (f(Y) * (0 * f(s(Y))))$. One can easily verify that

$$\begin{array}{rcl} \bar{\lambda}(t,\mathcal{T}) & = & \{(1,\mathrm{R}_4,id),(2{\cdot}1,\mathrm{R}_3,\{Y \mapsto s(s(Y_1))\}),(2{\cdot}2,\mathrm{R}_2,id),\\ & & (2{\cdot}2{\cdot}2,\mathrm{R}_3,\{Y \mapsto s(Y_2)\})\} \\ \bar{\bar{\lambda}}(t,\mathcal{T}) & = & \{id\} \end{array}$$

(for some *pdt* $\mathcal{T}$). The unifier $\{Y \mapsto s(s(Y_1))\}$ is discarded since it is an instance of $\{Y \mapsto s(Y_2)\}$. The unifier $\{Y \mapsto s(Y_2)\}$ is discarded since the redex created by its application is non-outermost. Thus the strategy $\bar{\bar{\lambda}}$ rewrites the term t in parallel at positions 1 and $2{\cdot}2$.

Theorem 4 $\overset{\bar{\bar{\lambda}}}{\rightsquigarrow}$ *is sound and complete in the sense of Theorems 2 and 3.*

6 Parallel Narrowing with Simplification

The strategy $\bar{\bar{\lambda}}$ improves weakly needed narrowing, but it may still perform some redundant computations, as shown in the following example.

[4] The set notation $\{\sigma_{|\mathcal{V}ar(t)} \mid \cdots\} \,/ \equiv$ means that this set must not contain two substitutions σ_1, σ_2 with $\sigma_1 \equiv_{\mathcal{V}ar(t)} \sigma_2$.

Example 7 Consider the rules of Example 6. Let $t = f(s(Y)) * f(s(s(Y)))$. Then, for an appropriate *pdt* $\mathcal{T}$, $\bar{\bar{\lambda}}(t, \mathcal{T}) = \{id, \{Y \mapsto s(Y_2)\}\}$. If we develop the search space of t, we will compute twice the result 0 with the substitution id and the redundant substitution $\{Y \mapsto s(Y_2)\}$. However, if we simplify t to $f(s(Y)) * 0$ by applying a rewrite step with rule R_3 (note that all rules except R_4 are terminating) before applying a parallel narrowing step, we will compute only once the result 0 with the identity substitution.

In this section we define a new parallel narrowing strategy which combines the strategy $\bar{\bar{\lambda}}$ with a kind of term simplification. The resulting strategy is complete and avoids some useless computations performed by $\bar{\bar{\lambda}}$. In order to support flexible simplification strategies, we combine $\bar{\bar{\lambda}}$ with a *simplifying rewriting strategy* which is a mapping $\mathcal{S}$ from terms to terms such that

1. $\forall\, t \in \mathcal{T}(\Sigma, \mathcal{X}), \mathcal{S}(t) = t' \Rightarrow t \xrightarrow{*} t'$ (i.e., $\mathcal{S}$ is compatible with rewriting)
2. $\mathcal{S}$ is recursive (i.e., $\mathcal{S}$ is computable).

For instance, mapping a term to itself, or its reduct, or one of its descendants obtained using terminating rules are all plausible simplifying rewriting strategies. The following definition introduces a new parallel narrowing strategy that combines $\bar{\bar{\lambda}}$ and a simplifying rewriting strategy. We denote by $\bar{\bar{\lambda}}(t)$ the substitution set $\bar{\bar{\lambda}}(t, \mathcal{T})$ if t is operation-rooted and $\mathcal{T}$ is a parallel definitional tree of the root of t, or the empty set if t is not operation-rooted.

Definition 7 Let $\mathcal{R}$ be a CAT, $\mathcal{S}$ a simplifying rewriting strategy, and t an operation-rooted term. We call *parallel narrowing with simplification* the binary relation $\bar{\bar{\lambda}}_{\mathcal{S}}$ over terms defined as follows: $t \overset{\bar{\bar{\lambda}}_{\mathcal{S}}}{\leadsto}_{\sigma} t'$ iff either

- $\sigma \in \bar{\bar{\lambda}}(\mathcal{S}(t))$ and $\mathcal{S}(t) \overset{\bar{\bar{\lambda}}}{\leadsto}_{\sigma} t'$, or
- $\bar{\bar{\lambda}}(\mathcal{S}(t)) = \varnothing$, $t' = \mathcal{S}(t)$, $t' \neq t$, and $\sigma = id$.

Thus, parallel narrowing with simplification deterministically simplifies a term before applying a narrowing step. It may happen that no narrowing step is applicable after simplification since the term may be reduced to normal form, which is the reason for the second case in the definition.

Theorem 5 *Let $\mathcal{S}$ be a simplifying rewriting strategy. The parallel narrowing with simplification strategy $\bar{\bar{\lambda}}_{\mathcal{S}}$ is sound and complete in the sense of Theorems 2 and 3.*

If we use a parallel rewriting strategy similar to $\bar{\varphi}$ to compute simplification steps, then the simplification steps can also be considered as narrowing steps where the applied substitution is the identity. Therefore, one might suppose that the commitment to the identity substitution in the definition of $\bar{\bar{\lambda}}$ (whenever possible) has the same effect as simplification. Unfortunately, such a commitment destroys the completeness of parallel narrowing, as can be seen by developing the search space for the term $g(X) * f(Y)$ w.r.t. the rules in Example 6.

7 Optimality

In this section we discuss two optimality results of our narrowing strategies. Inductively sequential systems are a subclass of CATs. An inductively sequential operation f has a parallel definitional tree $\mathcal{T}$ with exactly one sequential component, i.e., $\mathcal{T}$ itself is a (sequential) definitional tree. Both weakly needed narrowing and parallel narrowing behave as needed narrowing when they operate on such a tree.

Theorem 6 *Let $\mathcal{R}$ be a CAT, t an operation-rooted term whose defined operations are all inductively sequential. Then, for appropriate definitional trees for the operations in t, the narrowing steps of t computed by both weakly needed narrowing and parallel narrowing are the same as the narrowing steps of t computed by needed narrowing.*

We now turn our attention to the behavior of parallel narrowing on ground goals.

Theorem 7 *The parallel narrowing strategy is (deterministically) normalizing on ground goals.*

The above results show that parallel narrowing is a conservative extension of two optimal strategies, needed narrowing on inductively sequential systems and rewriting necessary sets on ground terms.

The strong optimality results of needed narrowing cannot be expected to hold for both weakly needed and parallel narrowing. In particular, we recall that rewriting and/or narrowing needed positions is not always possible in almost orthogonal TRSs, since such positions generally do not exist [21]. Furthermore, computing only independent unifiers seems unlikely, too, without look-ahead, as the next example shows.

Example 8 Consider the parallel-or of Example 2 together with the rules

$$f(0,X) \rightarrow X \qquad\qquad h(0) \rightarrow true$$

and the goal $f(X,h(Y)) \vee f(Y,h(X))$. Parallel narrowing computes two derivations of this goal beginning with different unifiers, eventually to discover that they yield the same substitution.

8 Related Work

In this section we compare our parallel narrowing strategy with other narrowing strategies proposed for CATs. There are also many narrowing strategies for other rewrite systems than CATs, like innermost, outermost, or basic narrowing (see [10]). However, all these strategies require the termination of the rewrite relation which is an undecidable property and immediately excludes typical functional programming techniques like infinite data structures. To ensure confluence in the presence of non-terminating rules, weak orthogonality and constructor-based rewrite rules are natural requirements. For this class of rewrite systems, lazy narrowing has been proposed (see, e.g., [7, 18, 20]). Similarly to lazy evaluation in functional languages, lazy narrowing evaluates an inner term only when its value is demanded to narrow an outer term. In contrast to functional languages, a naive version of lazy narrowing may evaluate the same argument several times due to the non-deterministic choice of a function's rewrite rules. Therefore, several methods have been proposed aiming at evaluating arguments commonly demanded by all rules before the non-deterministic choice (e.g., [2, 16]). Needed narrowing [2] is the only strategy that has been shown to be optimal w.r.t. the length of derivations and the number of computed solutions. Needed narrowing is defined for inductively sequential systems, and we have shown in Theorem 6 that parallel narrowing is a conservative extension of this optimal strategy.

In case of overlapping rules, the situation is more difficult since an argument may be demanded by some rule but not demanded by another rule for the same function. This has the unfortunate effect that naive lazy narrowing is often inefficient for such rules [9]. There are different proposals to improve naive lazy narrowing in this case. For instance, Loogen and Winkler [17] propose the *dynamic cut* which ignores subsequent alternative rules for narrowing if a rule is applicable without binding of goal variables. The effect of the dynamic cut is subsumed by our strategy since parallel narrowing prefers deterministic reductions at the root:

Proposition 1 *Let $\mathcal{R}$ be a CAT, t an operation-rooted term and $l \to r \in \mathcal{R}$ a rule with $\sigma(l) = t$ for some substitution σ. Then $t \overset{\bar{\bar{\lambda}}}{\leadsto}_{id} \sigma(r)$ is the only parallel narrowing step starting at t.*

This proposition also shows another advantage of our parallel narrowing strategy in comparison to the dynamic cut: parallel narrowing is independent of the order of rewrite rules. Since the dynamic cut only discards alternative rules *after* the current rule, it has no effect if the application of previous rules instantiates variables. This disadvantage is omitted in [9] where the combination of lazy narrowing with possible reduction steps between narrowing steps is proposed. In order to ensure the completeness of this *lazy narrowing with simplification* strategy, a terminating subset of all rewrite rules is used for reduction. Parallel narrowing does not subsume lazy narrowing with simplification, as can be seen in Example 7. However, simplification with a terminating set of rewrite rules is a simplifying rewrite strategy. Therefore, parallel narrowing with simplification has the same advantage as lazy narrowing with simplification.

Parallel narrowing is not intended as a strategy to implement functional logic languages on parallel architectures due to its fine-grained parallelism. This is in contrast to the AND-parallel narrowing implementation presented in [15] where independent subterms are evaluated in parallel. However, due to the fact that parallel narrowing reduces the number of non-deterministic choices in narrowing steps (compared to classic narrowing), parallel narrowing is useful to avoid redundant computations in OR-parallel implementations of narrowing.

The following table summarizes the characteristics of the major narrowing strategies for weakly orthogonal constructor-based rewrite systems. "Ground deterministic" means that a strategy performs only deterministic computations steps for all programs and all ground goals. "Normalizing" is satisfied if a strategy computes the normal form of a (non-ground) goal G satisfying

$G \overset{*}{\to} true$ in a fully deterministic way. In this case, a sequential implementation of this strategy always computes the normal form whenever it exists.

Strategy	Ground deterministic:	Normalizing:	Optimality properties:
simple lazy [18, 20]	no	no	
weakly needed [16]	no	no	
dynamic cut [17]	no	no	
lazy narrowing with simplification [9]	terminating TRS	terminating TRS	
parallel narrowing	yes	no	inductively sequential TRSs: shortest derivations (sharing) minimal number of solutions
parallel narrowing with simplification	yes	terminating TRS	

Parallel narrowing (with simplification) is deterministic on ground terms by Theorem 7. However, parallel narrowing without simplification is not normalizing as shown in Example 7. The optimality of parallel narrowing follows from the optimality results for needed narrowing [2] by Theorem 6.

This table shows that parallel narrowing is not only a further narrowing strategy with some optimizations, but it is the only strategy which subsumes the advantages of known lazy narrowing strategies together with clearly defined optimality results. Due to its fully deterministic behavior on functional programs (ground terms) and its ability to compute solutions to non-ground goals, it is the first sound and complete strategy which combines the evaluation mechanisms of functional and logic programs in a seamless way.

9 Conclusions

We have presented a new narrowing strategy for weakly orthogonal, constructor-based rewrite systems. Since this class includes non-terminating systems, it adequately models the functional component of modern, integrated functional logic languages. The main idea of our narrowing strategy is the parallel evaluation of necessary sets of redexes. This leads to a generalization of Sekar and Ramakrishnan's work on rewriting to narrowing. Parallel narrowing is a conservative extension of an optimal narrowing strategy, needed narrowing [2], to weakly orthogonal rewrite systems. Furthermore, parallel narrowing is the only known narrowing strategy for possibly non-terminating and overlapping TRSs which evaluates ground terms in a fully deterministic way. It can be implemented relatively efficiently, since narrowing steps are computed by local computations based on unification.[5] These features seem to make this strategy the best available choice for the implementation of functional logic programming languages.

Acknowledgements. Sergio Antoy was supported in part by the National Science Foundation under grant CCR-9406751. Rachid Echahed was supported in part by the French Centre National de la Recherche Scientifique (GDR Programmation du CNRS) and by Portland State University. 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).

[5] An implementation of parallel narrowing based on the compilation into Prolog is described in [6].

References

[1] S. Antoy. Definitional trees. In *Proc. of the 4th Intl. Conf. on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.

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

[3] S. Antoy, R. Echahed, and M. Hanus. Parallel Evaluation Strategies for Functional Logic Languages Portland State University, 1996. Available via URL ftp://ftp.cs.pdx.edu/pub/faculty/antoy/Parallel-Evaluation-Strategies.ps.Z

[4] 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.

[5] 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.

[6] D. Genius. Sequential implementation of parallel narrowing. In *Proc. JICSLP'96 Workshop on Multi-Paradigm Logic Programming*, pages 95–104. TU Berlin, Technical Report No. 96-28, 1996.

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

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

[9] M. Hanus. Combining lazy narrowing and simplification. In *Proc. of the 6th Intl. Symp. on Programming Language Implementation and Logic Programming*, pages 370–384. Springer LNCS 844, 1994.

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

[11] 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, 1991.

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

[13] 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.

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

[15] H. Kuchen, J.J. Moreno-Navarro, and M.V. Hermenegildo. Independent and-parallel implementation of narrowing. In *Proc. of the 4th Intl. Symp. on Programming Language Implementation and Logic Programming*, pages 24–38. Springer LNCS 631, 1992.

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

[17] R. Loogen and S. Winkler. Dynamic detection of determinism in functional logic languages. *Theoretical Computer Science 142*, pages 59–87, 1995.

[18] 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.

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

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

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