Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Symbolic Computation 40 (2005) 875–903
Journal of Symbolic Computation
www.elsevier.com/locate/jsc

Evaluation strategies for functional logic programming

Sergio Antoy*

Computer Science Department, Portland State University, P.O. Box 751, Portland, OR 97207, USA

Received 13 January 2003; accepted 14 December 2004
Available online 2 March 2005

Abstract

Recent advances in the foundations and the implementations of functional logic programming languages originate from far-reaching results on narrowing evaluation strategies. Narrowing is a computation similar to rewriting which yields substitutions in addition to normal forms. In functional logic programming, the classes of rewrite systems to which narrowing is applied are, for the most part, subclasses of the constructor-based, possibly conditional, rewrite systems. Many interesting narrowing strategies, particularly for the smallest subclasses of the constructor-based rewrite systems, are generalizations of well-known rewrite strategies. However, some strategies for larger non-confluent subclasses have been developed just for functional logic computations. This paper discusses the elements that play a relevant role in evaluation strategies for functional logic computations, describes some important classes of rewrite systems that model functional logic programs, shows examples of the differences in expressiveness provided by these classes, and reviews the characteristics of narrowing strategies proposed for each class of rewrite systems.

Keywords: Evaluation strategies; Narrowing; Definitional trees; Constructor-based rewrite systems; Functional logic programming

* Tel.: +1 503 725 3009; fax: +1 503 725 3211.
E-mail address: antoy@cs.pdx.edu.

0747-7171/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jsc.2004.12.007

1. Introduction

Functional logic programming studies programming languages that combine the distinctive features of functional programming (algebraic data types, lazy evaluation, polymorphic typing, first-class functions, monadic I/O) and logic programming (logic variables, non-determinism, built-in search). A substantial problem of combining these paradigms is that when executing a program it may be necessary to evaluate a functional-like expression containing uninstantiated logic variables. Two operational principles have been proposed for this situation, *residuation* and *narrowing*; see Hanus (1994) for a survey. In short, residuation delays the evaluation of expressions containing uninstantiated logic variables, whereas narrowing guesses an instantiation for these variables. Functional logic languages can be effectively implemented with either operational principle: for example, Life (Aït-Kaci, 1990) and Escher (Lloyd, 1999) are based on residuation, Babel (Moreno-Navarro and Rodríguez-Artalejo, 1992) and $\mathcal{TOY}$ (López-Fraguas and Sánchez-Hernández, 1999) are based on narrowing, and Curry (Hanus, 2003) supports both residuation and narrowing. This paper is about narrowing, in particular about narrowing strategies for functional logic computations.

Functional logic programs can be quite expressive; see Example 1. The language used to present the examples is abstract to avoid the details associated to concrete practical languages. The syntax is inspired by Curry, which in turn is an extension of Haskell (Peyton Jones and Hughes, 1999). A brief explanation of the syntactic conventions used in this paper will follow shortly.

Example 1. Functional logic program for the problem of the *Dutch National Flag* (Dijkstra, 1976): given a sequence of pebbles, each having one of the colors red, white and blue, rearrange the pebbles so that they appear in the order of the Dutch flag.

```
solve FLAG -> solve (X ++ [red | Y] ++ [white | Z])
           :- FLAG = X ++ [white | Y] ++ [red | Z]
solve FLAG -> solve (X ++ [red | Y] ++ [blue | Z])
           :- FLAG = X ++ [blue | Y] ++ [red | Z]
solve FLAG -> solve (X ++ [white | Y] ++ [blue | Z])
           :- FLAG = X ++ [blue | Y] ++ [white | Z]
solve FLAG -> FLAG
           :- FLAG = uni red ++ uni white ++ uni blue

uni COLOR -> []
uni COLOR -> [COLOR | uni COLOR]
```

For the most part, a functional logic program can be seen as a constructor-based conditional rewrite system (TRS). In presenting the examples, I deviate a little from the standard notation of TRSs. This deviation keeps the examples smaller, closer to real programs and easier to understand.

TRSs are first-order languages, but in this paper the notation for function and constructor application is curried as usual in functional programming. A conditional rewrite rule has the form

$$l \to r :- t_1 = u_1, \ldots, t_n = u_n$$

where l and r are the left- and right-hand sides, respectively, and the condition is a sequence of elementary equational constraints of the form $t_i = u_i$. The meaning of the symbol "=" in programming languages is slightly more restrictive than that in rewriting. The difference will be discussed in some detail later.

The program of Example 1 adopts the familiar Prolog notation for both lists and variables and uses common infix operators, but this is only syntactic sugar. The identifier "++" stands for list concatenation and is right associative. The *operation* `uni` returns a list of pebbles all of the color of its argument. A list of any length can be returned by this operation. The operation `solve` repeatedly swaps pebbles until the problem is solved. Any pair of pebbles out of place with respect to the flag colors can be swapped. The program is executed by replacing an instance of a rule's left-hand side with the corresponding instance of the rule's right-hand side, provided that the rule's condition is satisfied. A free variable in a condition, i.e., `X`, `Y` and `Z`, may be instantiated if its instantiation is useful for satisfying the condition.

The identifiers `uni` and `solve` are referred to as *operation* rather than *function* symbols. These symbols do not identify functions in the ordinary mathematical sense. For example, the application of `uni` non-deterministically returns distinct results for the same argument. The word "operation" may be preferable to highlight this characteristic. However, the words "function" and "non-deterministic function" are often used, too, to highlight that these symbols can be used as ordinary function symbols in a functional logic program. In particular, an application of these symbols can be functionally nested as for ordinary function symbols. Antoy (1997) shows that functional nesting is crucial for ensuring the lazy evaluation of expressions and consequently the completeness of functional logic computations.

To understand the promise of functional logic programming languages, I compare the above program with published examples of programs in declarative languages proposed for the same problem.

These programs are specified, in natural language, in forms that differ from Dijkstra's original formulation of the problem (Dijkstra, 1976). The *specification* of each program already provides some clues as to what one will find in the corresponding implementation. It is somewhat intended that the pebbles have an identity. Otherwise, counting how many pebbles of each color are present in the input would lead to a simple and efficient solution. Dijkstra's formulation (Dijkstra, 1976), in natural language, describes *two* "computer-controlled hands" that pick up two pebbles and swap them. This is all and only what the program of Example 1 does when two pebbles of whatever color and in whatever position are out of place for the flag. Both a rewrite system-based specification (Dershowitz, 1995) of this problem and an executable specification in the *ASF+SDF* meta-environment (Brand and Klint, 2003) swap exclusively *adjacent* pebbles out of place for the flag. Although I can only conjecture, this deviation from the original formulation is due to the fact that narrowing was not contemplated for the program execution.

The specifications of both the pure logic (O'Keefe, 1990) and the pure functional (Petersson and Smith, 1986) programs describe the problem as computing a *permutation* of the pebbles which is *sorted* according to the flag colors. The corresponding implementations can be summarized as *filtering* the pebbles of each color and *concatenating* the results. It appears that the cart is leading the horse. The specifiers took a considerable liberty. The spirit of the original, somewhat anthropomorphic, formulation with a "movable eye" and "two hands" that swap pebbles has been sacrificed to the programming language paradigms.

The "special requirements" of the original specification (Dijkstra, 1976) concerning the efficiency of the execution in an imperative language are largely ignored by these declarative programs, which all execute in the blink of an eye. Indeed, after a quarter of a century, the focus of a non-negligible portion of computer science has shifted from saving memory bits and CPU cycles to producing programs that are easy to code, to prove correct, to maintain and to understand, perhaps at the expense of an acceptable loss of efficiency. Declarative languages and functional logic languages in particular have a promising potential in this respect.

The functional logic program is textually shorter, closer to the specification and conceptually simpler than all the other alternatives. Key factors that contribute to this simplicity and are unavailable in either the functional or the logic program, or both, are: (1) *non-determinism*, e.g., the operation `solve` swaps any of the possibly many pairs of pebbles that can be out of place; (2) *semantic unification*, e.g., the variables `X`, `Y` and `Z` in the equations of the rules, e.g., `FLAG = X ++ [white | Y] ++ [red | Z]` in the first rule, are instantiated, if possible, to solve the equation; (3) *functional inversion*, i.e., the value of some argument(s) of a function is computed from the function's result, e.g., the above equation is solved to split a list into sublists rather than to concatenate sublists into a result; and (4) *functional nesting* and *lazy evaluation*, e.g., in the above equation the subexpression `[red | Z]` is evaluated only after it is recognized that, for suitable values of `X` and `Y`, the subexpression `X ++ [white | Y]` is a prefix of `FLAG`.

Non-determinism, semantic unification, functional inversion, functional nesting and lazy evaluation, which are crucial for the expressiveness of functional logic programs, are supported by two specific aspects of functional logic computations: (1) modern functional logic programs are mostly executed by *narrowing*, a computation that generalizes both ordinary functional evaluation and resolution; and (2) the classes of TRSs modeling functional logic programs are more general than those modeling functional programs, e.g., our initial example includes both non-deterministic operations, such as `solve` and `uni`, and extra variables, such as `X`, `Y` and `Z` in some rules of `solve`.

It is well known that certain functional computations can be implemented in a logic programming language with a technique referred to as *flattening* (Bosco et al., 1988). Higher-order functions can be accommodated as well (Warren, 1982). With this technique, a functional program is transformed, or flattened, into a logic program intended to compute the same input/output function. Functional logic programs can be flattened as well since the resolution-based computation of the logic program simulates or implements the narrowing-based computation of the original functional logic program. However, some functional logic programs do not behave as intended when flattened. In particular, both non-determinism and non-termination are crucial factors.

The program of Example 1, when flattened in Prolog, fails to solve the problem for some very simple inputs, e.g., `[blue,red]`. The analysis of its execution shows that the reason is the depth-first search strategy of the Prolog computation model. The first rule of the program generates an infinite search space that prevents any attempt to execute other rules. However, the size of this search space is unmotivated. The program of Example 1 can be coded in Curry with changes that are mostly syntactic. This program is executed as intended for any input by the PAKCS compiler–interpreter (Hanus et al., 2003), which translates source Curry code into source Prolog code using the technique described in Antoy and Hanus (2000). The resulting Prolog code computes with the depth-first search strategy. The reason that this program generates a finite search space is that flattening removes functional nesting and consequently the possibility of evaluating expressions lazily. Example 14, from Antoy (1997), shows in a much simpler situation that functional nesting and lazy evaluation are essential for the intended as well as the technical completeness of functional logic computations.

This paper discusses and compares several key aspects of the evaluation of functional logic computations. The main contribution is a survey of four classes of TRSs. For each class, the paper presents a narrowing strategy for the execution of computations in that class. The paper also recalls the notion of *definitional tree* (Antoy, 1992) which ultimately supports each presented strategy.

Section 2 reviews narrowing as the computation of functional logic programs. Section 3 defines and compares various fundamental classes of TRSs proposed to model functional logic programs and, for each class, presents an evaluation strategy. Section 4 briefly discusses some extensions to the previous classes and related issues. Section 5 contains the conclusion.

2. Narrowing

This section briefly recalls basic notions of term rewriting (Baader and Nipkow, 1998; Dershowitz and Jouannaud, 1990; Klop, 1992) and functional logic programming (Hanus, 1994).

A *rewrite system* is a pair, $\mathcal{R} = \langle \Sigma, R \rangle$, where Σ is a *signature* and R is a set of *rewrite rules*. The signature Σ is *many-sorted* and is partitioned into a set $\mathcal{C}$ of *constructor* symbols and a set $\mathcal{F}$ of *defined operations* or functions. $\text{TERM}(\Sigma \cup \mathcal{X})$ is the set of *terms* constructed over Σ and a countably infinite set $\mathcal{X}$ of *variables*. $\text{TERM}(\mathcal{C} \cup \mathcal{X})$ is the set of *values*, i.e., the set of terms constructed over $\mathcal{C}$ and $\mathcal{X}$. $\mathcal{V}ar(t)$ is the set of the variables occurring in a term t. Terms are well-typed.

A *pattern* is a term of the form $f(t_1, \ldots, t_n)$, $n \geqslant 0$, where f is a function, or operation, of arity n and $t_1, \ldots, t_n$ are values. An *unconditional rewrite rule* is a pair $l \to r$, where l is a linear pattern and r is a term. *Linear* means that repeated occurrences of the same variable in l are not allowed. This restriction will be justified later. Traditionally, it is required that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. This condition limits the expressiveness of functional logic programming languages and it is generally relaxed in implementations, although some important results of the theory of narrowing strategies have been developed with this condition. A substitution is an idempotent mapping $\sigma : \mathcal{X} \to \text{TERM}(\mathcal{C} \cup \mathcal{X})$, implicitly

extended to terms, such that the domain of σ, $\mathrm{dom}(\sigma) = \{x \in \mathcal{X} | \sigma(x) \neq x\}$, is finite. An unconditional TRS, $\mathcal{R}$, defines a rewrite relation $\rightarrow_{\mathcal{R}}$ on terms as follows: $s \rightarrow_{p,R} t$ if there exists a position p in s, a rewrite rule $R = l \rightarrow r$ with fresh variables and a substitution σ such that $s|_p = \sigma(l)$ and $t = s[\sigma(r)]_p$. The instantiated left-hand side $\sigma(l)$ of a rewrite rule $l \rightarrow r$ is called a *redex* (*red*ucible *ex*pression). Given a relation $\rightarrow$, $\overset{+}{\rightarrow}$ and $\overset{*}{\rightarrow}$ denote its transitive closure and its transitive and reflexive closure, respectively.

A *conditional* rewrite rule has the form $l \rightarrow r :- c$, where l and r are defined as in the unconditional case and c is a possibly empty *sequence of elementary equational constraints*, i.e., pairs of terms of the form $t = u$. The definition of the rewrite relation for conditional TRSs (Bergstra and Klop, 1986), see also Bezem et al. (2003, Sect. 3.5), is more complicated than for unconditional TRSs. Intuitively, $s \rightarrow_{p,R} t$ if there exists a position p in s, a rewrite rule $R = l \rightarrow r :- c$ with fresh variables and a substitution σ such that $s|_p = \sigma(l)$, $\sigma(c)$ holds and $t = s[\sigma(r)]_p$. There is an apparent or potential circularity in this intuitive statement since to satisfy the instantiated condition, $\sigma(c)$, one refers to the rewrite relation being defined. This problem is resolved by an inductive definition. For the base case, only syntactically equal terms are in the rewrite relation, i.e., $s \overset{*}{\rightarrow} t$ iff $s \equiv t$. For the induction case, if every elementary equational constraint of $\sigma(c)$ is in the rewrite relation, then $s \overset{*}{\rightarrow} t$, i.e., $\langle s, t \rangle$ is in the rewrite relation. In rewriting, the meaning of the symbol "=" is *convertibility*, i.e., $t = u$ if and only if t can be converted into u (and vice versa) by equational reasoning.

A left-linear, conditional, constructor-based TRS is a good model for a functional or a functional logic program. A computation is (abstracted by) an operation-rooted term. A result is a value, i.e., a constructor term. In functional logic programming the symbol "=" is referred to as *strict equality* (Giovannetti et al., 1991; Moreno-Navarro and Rodríguez-Artalejo, 1992). Its meaning, as the name suggests, is stricter than in rewriting. A justification of this decision is given in Example 19. In functional and functional logic programming, $t = u$ if and only if t and u can be evaluated to the same value.

Example 2. In programming languages, values are introduced by data type declarations such as:

```
data color  = white | red | blue
data list a = [] | [a | list a]
```

and operations are defined by rewrite rules such as those of Example 1. The identifiers `white`, `red` and `blue` are (constant) data constructors of the type `color`. The constructors of the polymorphic type `list` are `[]` (empty list) and `[·|·]` (non-empty list). The identifier `a` is a type variable ranging over all types. A value or *data term* is a well-formed expression containing variables and data constructors, e.g., `[red,blue]` which stands for `[red|[blue|[]]]`.

The fundamental computation mechanism of functional logic languages is *narrowing*. A term s *narrows* to t with substitution σ, denoted $s \leadsto_{p,R,\sigma} t$, if p is a non-variable position of s, R is a rewrite rule, and σ is a substitution such that $\sigma(s) \rightarrow_{p,R} t$. As defined earlier only idempotent constructor substitutions are considered in this paper. A term s

such that $\sigma(s)$ is a redex is called a *narrex* (*narr*owable *ex*pression). When σ is the identity substitution, a narrowing step becomes a rewrite step and a narrex becomes a redex.

Traditionally, it is required that the substitution of a narrowing step is a most general unifier of a narrex and a rule's left-hand side. This condition is not imposed here since narrowing with most general unifiers is suboptimal (Antoy et al., 2000). This will be further addressed later. A *computation* or *evaluation* of a term s is a narrowing derivation $s = t_0 \leadsto_{\sigma_1} \cdots \leadsto_{\sigma_n} t_n = t$, where t is a value. The substitution $\sigma_1 \circ \cdots \circ \sigma_n$ is called a *computed answer* and t is called a *computed value* of s. Computing narrowing steps, in particular narrexes and their substitutions, is the task of a *strategy*.

Example 3. The following rewrite rules define the concatenation and the strict equality of a polymorphic type list. The infix operation "`&`" is the constraint conjunction. The identifier `success` denotes a solved constraint. It is declared as the constructor of a singleton type not further identified here. It is explicitly represented in this paper to present computations strictly using only rewrite rules, but with appropriate conventions it could be eliminated from the concrete syntax of a language.

```
[] ++ X -> X                          R1
[X|Y] ++ Z -> [X | Y++Z]              R2
[] = [] -> success                    R3
[X|Xs] = [Y|Ys] -> X=Y & Xs=Ys        R4
success & X -> X                      R5
```

The definition of equality presented above leads to a poor operational behavior because if s and t are variables, solving $s = t$ requires grounding the variables. To alleviate this problem, the run-time systems of some functional logic languages provide the strict equality as a built-in, ad hoc operation that in the above situation unifies the variables. Within the framework of rewriting, formalizing this behavior is hard at best. Narrowing calculi, e.g., see González Moreno et al. (1999); Middeldorp and Okui (1998), are better suited for this task.

Example 4. The execution of the program of Example 1 requires the solution of constraints, such as `U++V=[red,white,red]`, which are solved by narrowing. A free variable may have different instantiations. Consequently, expressions containing free variables may be narrowed to different results. Below is the initial portion of one of several possible sequences of steps that solve the constraint, i.e., narrow it to `success` and in the process instantiate the variables `U` and `V`. Both the rule and the substitution applied in a step are shown to the right of the reduct:

```
U++V=[red,white,red]
      ⇝ [U1|Us++V]=[red,white,red]      R2, {U ↦ [U1|Us]}
      ⇝ U1=red & Us++V=[white,red]      R4, {}
      ⇝ success & Us++V=[white,red]     Ri, {U1 ↦ red}
      ⇝ Us++V=[white,red]               R5, {}
       ⋮
```

where $\mathtt{U}_1$ and $\mathtt{U}_s$ are fresh variables, and R_i denotes some rule, not shown here, of the strict equality of the type `color`. This solution instantiates the variable `U` to a list with head `red`.

A narrowing strategy is a crucial component of the foundations and the implementation of a functional logic programming language. Its task is the computation of the step, or steps, that must be applied to a term. In a constructor-based TRS, a narrowing step of a term t is identified by a non-variable position p of t, a rewrite rule $l \to r$, and an idempotent constructor substitution σ such that $t \leadsto_{p,l\to r,\sigma} s$ iff $s = \sigma(t[r]_p)$. Formally, a narrowing strategy is a mapping that takes a term t and yields one or more triples of the form $\langle p, l \to r, \sigma \rangle$ interpreted as narrowing steps as defined earlier.

Example 5. Continuing Example 3, a narrowing strategy (such as needed narrowing discussed later) applied to the constraint `U++V=[red,white,red]` computes two steps: $\langle 1, R_1, \{\mathtt{U} \mapsto \mathtt{[]}\}\rangle$ and $\langle 1, R_2, \{\mathtt{U} \mapsto \mathtt{[U_1|U_s]}\}\rangle$. The first step yields a solution with answer `U=[]` and `V=[red,white,red]`. The second step was shown earlier.

It should be obvious that narrowing extends rewriting by instantiating variables in addition to computing normal forms. For this reason, extra variables in rewrite rules are not only permitted, but they seem a natural element with which to compute. An *extra* variable v is a variable that occurs in the condition and/or the right-hand side of a rewrite rule R, but not in the left-hand side. The strategies discussed in the paper apply without problems to rewrite rules with extra variables, but their most important properties have been proved for rules without extra variables. More details will be provided later. Consider again the first rewrite rule of operation `solve` in Example 1. The variables `X`, `Y` and `Z` are extra variables of this rule. Given the program of this example and some sequence of colors, `FLAG`, the program can evaluate the expression:

```
FLAG = X ++ [white | Y] ++ [red | Z]
```

where `X`, `Y` and `Z` are unbound. For `FLAG=[red,white,red]` this expression eventually evaluates to `success` and binds `X` to `[red]` and both `Y` and `Z` to the empty list.

A narrowing strategy useful for functional logic programming must be *sound*, *complete* and *efficient*. In the next definitions, t and u denote a term and a value, respectively, and all narrowing derivations are computed by the strategy subject of the discussion. A strategy is *sound* iff $t \overset{*}{\leadsto}_\sigma u$ implies $\sigma(t) \overset{*}{\to} u$. A strategy is *complete* iff $\sigma(t) \overset{*}{\to} u$ implies the existence of a substitution $\eta \leqslant \sigma$ such that $t \overset{*}{\leadsto}_\eta u'$, where $u = \rho(u')$, for some idempotent constructor substitution ρ. In practice, only the substitution of the variables of t is interesting. Both the soundness and the completeness of a strategy have an intuitive explanation when the initial term of a derivation is an equation containing unknown variables. In this case, the soundness of a strategy guarantees that any instantiation of the variables computed by the strategy is a solution of the equation, and the completeness guarantees that for any solution of the equation, the strategy computes another solution which is at least as general.

Efficiency is a more elusive property. In practice, it is desirable to minimize the overall time and memory consumed to find one or all the values of an expression. This

is somewhat related to the length of the derivations and even more to the size of the search space, although reasoning about the latter seems very difficult. In any case, two factors clearly affecting the efficiency of a strategy are: (1) unnecessary steps should be avoided; and (2) steps should be computed without consuming unnecessary resources. In both statements, the exact meaning of "unnecessary" is difficult to formalize at best. Factor (1) is more related to the theory of a strategy, whereas factor (2) is more related to its implementation, although the boundaries of these relationships are blurred. The efficiency of a strategy is somewhat at odds with its completeness. A straightforward way to ensure the completeness of a strategy is to compute all possible narrowing steps of a term, but in most cases the strategy would be quite inefficient since many of these steps would be unnecessary.

Similar to the case for rewriting, different narrowing strategies have been proposed for different classes of TRSs. Some efficient narrowing strategies are extensions of corresponding rewrite strategies, whereas other narrowing strategies have been developed specifically for classes of TRSs well suited to functional logic programming and do not originate from previous rewrite strategies. Some of these classes and their strategies are the subject of the next section.

3. Classes of TRSs

A key decision in the design of functional logic languages is the class of TRSs chosen to model the programs. In principle, generality is very desirable since it contributes to the expressive power of a language. In practice, extreme power or the greatest generality are not always an advantage. The use of "unstructured" rewrite rules has two interrelated drawbacks: for the programmer it becomes harder to reason about the properties of a program; for the implementor it becomes harder to implement a language efficiently. For these reasons, different classes of TRSs potentially suitable for functional logic computations have been extensively investigated. Fig. 1 presents a containment diagram of some major classes. All the classes considered in the diagram are constructor-based. Rewrite rules defining an operation with the *constructor-discipline* (O'Donnell, 1977) implicitly define a corresponding function, possibly non-deterministic, over algebraic data types such as those of Example 2. Most often, this is well suited for programming, particularly when data types are abstract.

The discussion of this section is limited to first-order computations although higher-order functions are essential in functional, and hence functional logic, programming. The following section will address this limitation. The discussion of this section is also limited to unconditional TRSs. Conditional constructor-based TRSs can be transformed into unconditional TRSs by a transformation intended to preserve both values and computations without loss of either efficiency or generality. This also will be addressed in the next section. Finally, extra variables are excluded from rewrite rules since some important results of the theory of narrowing strategies have been proved for rules without extra variables. Where appropriate, the consequences that extra variables have on definitions, properties and computations will be discussed.

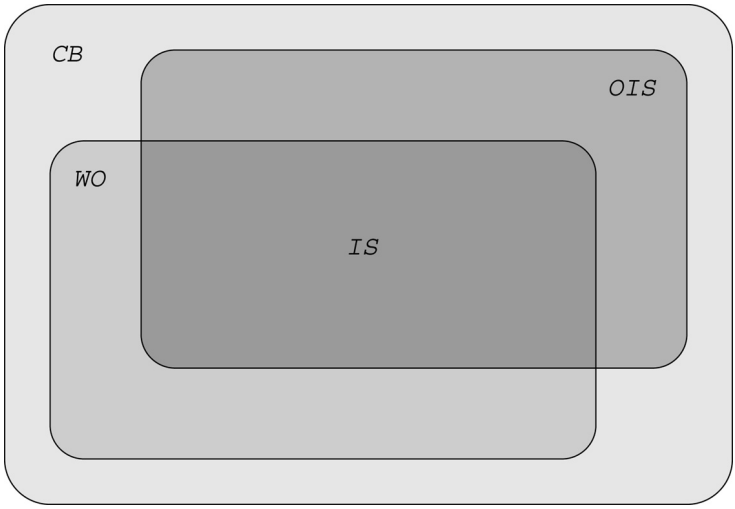


Fig. 1. Containment diagram of rewrite systems modeling functional logic programs. The outer area, labeled *CB*, represents the *constructor-based* rewrite systems. The smallest darkest area, labeled *IS*, represents the *inductively sequential* rewrite systems. These are the intersection of the *weakly orthogonal*, labeled *WO*, and the *overlapping inductively sequential* rewrite systems, labeled *OIS*.

3.1. Inductively sequential TRSs

The smallest class in the diagram of Fig. 1 is the *inductively sequential* TRSs. This class is important because it models the (first-order component of) programs of successful programming languages such as ML and Haskell. An efficient evaluation strategy with some remarkable properties is known for this class.

In orthogonal TRSs, every term t that is not in normal form has a *needed* redex s. Informally, the normal form of t cannot be reached unless s is contracted. The formalization of this claim is complicated since s could be affected if some other redex different from s is contracted in t. In general, s may be erased (of course, in this case it would not be *needed*), or it may change in the sense that some proper subredex of s may be contracted, or even several copies of s may be introduced in the reduct of t. Despite these possible outcomes, the identity of s can be traced through the rewrite steps of a computation using the notion of *descendant* (Huet and Lévy, 1991). Intuitively, one marks some symbol occurrences in t, e.g., the root symbol of s, by underlining or coloring it (Boudol, 1985), and looks for underlined or colored symbols in the reduct of t. Thus, a redex s of a term t is a *needed* redex if a descendant of s is contracted in any derivation of t to a normal form.

Needed redexes are uncomputable in the whole class of the orthogonal rewrite systems, but are computable in the smaller subclass of the strongly sequential rewrite systems. In this class, the *call-by-need* strategy (Huet and Lévy, 1991) repeatedly contracts a needed redex of a term and this suffices to reach a normal form, if it exists. This strategy is optimal in the sense that no rewrite step is wasted, since only redexes that sooner or later must be

contracted are contracted. However, the order in which redexes are contracted may affect the total number of steps executed to reach a normal form.

The inductively sequential TRSs can be characterized as the strongly sequential component of the constructor-based TRSs (Hanus et al., 1998). *Needed narrowing* (Antoy et al., 2000) is a conservative extension of the call-by-need strategy, i.e., rewrite derivations executed by needed narrowing are call-by-need derivations. Therefore, needed narrowing extends the optimality of the call-by-need strategy. In addition, needed narrowing offers a second optimality result concerning computed answers. Narrowing is non-deterministic; thus a term may have several distinct derivations each computing a substitution and a value. The substitutions computed by these derivations are pairwise disjoint (Antoy et al., 2000). This implies that every needed narrowing derivation computing a value is *needed* in the sense that the substitution computed by one derivation cannot be obtained by any other derivation.

Despite the similarities between needed narrowing and the call-by-need strategy, there exist some interesting differences. The main one is that there is no longer a good characterization of the notion of a needed redex. In orthogonal TRSs, a redex uniquely identifies a step of a rewrite computation, but a narrex does not.

Example 6. Consider the following declaration of the natural numbers, which for the purpose of this discussion are represented in Peano's notation, and the usual addition and "less than or equal to" operations:

```
data nat = 0 | s nat

0   + Y -> Y
s X + Y -> s (X + Y)

0   <= _   -> true
s X <= 0   -> false
s X <= s Y -> X <= Y
```

Consider the term t = `U<=0+0`, where `U` is an unbound variable. The subterm w = `0+0` of t is a redex; hence it is a narrex. Asking whether w is a *needed* narrex of t is not a meaningful question for a functional logic computation. It is easy to see that `U` must be instantiated to either `0` or `(s _)` to compute a value of t. If `U` is instantiated to `0`, w is not a needed redex of the instantiated term. However, if `U` is instantiated to `(s _)`, w is a needed redex of the instantiated term. For this reason, in discussing narrowing computations one talks of needed *steps* rather than redexes. Of course, for ground terms, a needed redex identifies a needed narrowing step and vice versa.

A second interesting difference between the call-by-need rewrite strategy and needed narrowing concerns the computation of a needed step. In a term whose normal form is not a value, i.e., it contains some defined operation, needed narrowing may fail to compute a needed redex. This may happen even in ground terms.

Example 7. Continuing Example 6, consider the following operation `f`:

```
f 0 -> 0
```

and the term $t =$ `f(s(f 0))`. The subterm `f 0` of t is a needed redex, but needed narrowing fails to compute it.

This characteristic of needed narrowing is referred to as a "blessing in disguise" in Antoy et al. (2000). It is easy to see that the normal form of t is `f(s 0)`. Normal forms containing an operation symbol, `f` in this case, represent failed computations in constructor-based TRSs. The early failure of needed narrowing may prevent wasting resources for computations that are doomed to fail. Needed narrowing is sound and complete for computations that end in a value, i.e., a constructor term. In constructor-based TRSs, computations that terminate with a normal form containing occurrences of operation symbols are considered failures or errors (Sekar and Ramakrishnan, 1993). In functional logic programming, operation symbols may be allowed in the result of a computation when they are not fully applied, but these terms are ultimately seen as values (Antoy and Tolmach, 1999).

Inductively sequential TRSs were initially characterized through the concept of a definitional tree (Antoy, 1992). Definitional trees have become the tool of choice for the formalization and implementation of narrowing strategies for several subclasses of the constructor-based TRSs. A *definitional tree* of an operation f is a finite non-empty set $\mathcal{T}$ of linear patterns partially ordered by subsumption and having the following properties up to renaming of variables:

- [leaves property] The maximal elements, referred to as the *leaves*, of $\mathcal{T}$ are all and only variants of the left-hand sides of the rules defining f. Non-maximal elements are referred to as *branches*.
- [root property] The minimum element, referred to as the *root*, of $\mathcal{T}$ is $f(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are fresh, distinct variables.
- [parent property] If π is a pattern of $\mathcal{T}$ different from the root, there exists in $\mathcal{T}$ a unique pattern π' strictly preceding π such that there exists no other pattern strictly between π and π'. π' is referred to as the *parent* of π and π as a *child* of π'.
- [induction property] All the children of a pattern π differ from each other only at a common position which is referred to as *inductive*. This is the position of a variable in π.

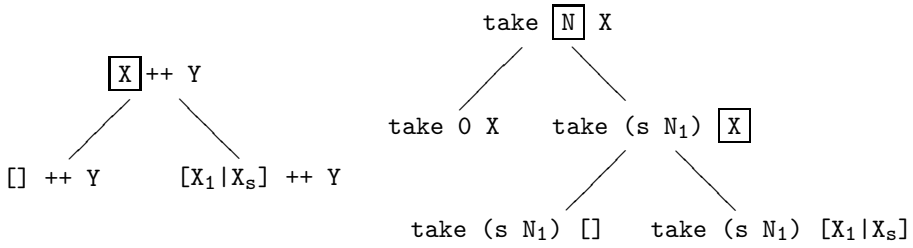
Since all the patterns of a definitional tree are linear, the names of the variables are irrelevant; hence all the variables could be anonymous. In the following examples, I give a name to variables to trace them from parent to children. This eases understanding how a definitional tree, and hence the definition of an operation, is obtained by a structural induction on a type.

Example 8. Consider an operation, `take`, that returns a prefix of a given length of a list:

```
take 0 _ -> []
take (s N) [] -> []
take (s N) [X|Xs] -> [X | take N Xs]
```

The definitional trees of operation "++", defined in Example 3, and operation `take` just defined are shown below. Lines join patterns in the parent–child relation. The inductive

variable of a parent is boxed. The leaves are variants of the rules' left-hand sides.



An operation is *inductively sequential* iff it has a definitional tree. A TRS is inductively sequential iff all its operations are inductively sequential. Needed narrowing is defined through definitional trees that are used as finite state automata to compute narrowing steps. An illustration of this computation is in Example 9. The formal definition is in Antoy et al. (2000).

Example 9. Needed narrowing computes a step of a term t rooted by `take`, i.e., $t =$ `take` $n\ x$, as follows. Let π be a maximal element of the definitional tree of `take` which unifies with t and let σ be the unifier. If π is a leaf, then t is a narrex and σ is the substitution of the step. If π is a branch and p is the position of its inductive variable, then $t|_p$ is rooted by some operation f. Using a definitional tree of f, the strategy computes a needed step of $\sigma(t|_p)$, say $\langle q, l \to r, \eta\rangle$. Then, $\langle p \cdot q, l \to r, \sigma \circ \eta\rangle$ is a needed step of t.

To make the example more concrete, suppose that $t =$ `take N ([1]++[2])`, where `N` is a free variable. The term t unifies with both `take 0 X`, which is a leaf, and `take (s` N_1`) X`, which is a branch. The first is obviously a maximal element in its tree, since it is a leaf. The second is maximal as well, since t does not unify with either of its children. Therefore, needed narrowing computes the two steps shown below. Each step is shown with its substitution:

$$\texttt{take N ([1]++[2])} \leadsto_{\texttt{N}\mapsto\texttt{0}} \texttt{[]}$$
$$\texttt{take N ([1]++[2])} \leadsto_{\texttt{N}\mapsto\texttt{(s N}_1\texttt{)}} \texttt{take (s N}_1\texttt{) [1|[]++[2]]}$$

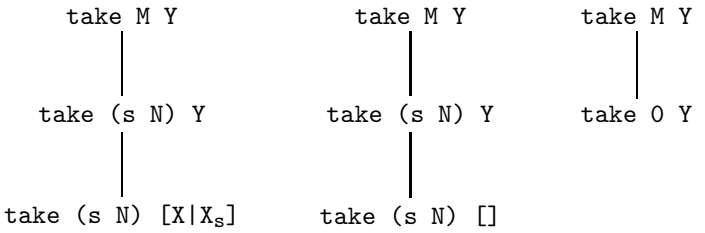
Observe that the substitution of the second step is not most general. This characteristic of needed narrowing is a major departure from previously proposed narrowing strategies. To understand why renouncing most general unifiers is an essential contribution to the strategy's optimality, consider a step with the same narrex and a most general unifier, i.e.:

$$\texttt{take N ([1]++[2])} \leadsto_{\{\}} \texttt{take N [1|[]++[2]]} \tag{1}$$

To compute a value of t, some step following (1) must instantiate `N` to either `0` or `(s` N_1`)`. When `N` is instantiated to `0`, it is easy to verify that t evaluates to `[]`, the empty list, and step (1) could have been entirely avoided.

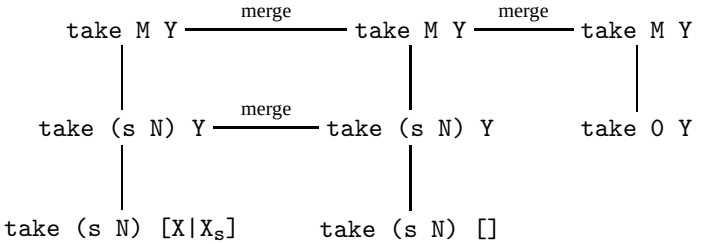
The inductive sequentiality of a TRS is a decidable property. A simple, elegant, nondeterministic algorithm for computing definitional trees is presented in Barry (1996). The algorithm has two main phases. In the first phase, a sequence of terms is obtained from the left-hand side of each rule of an operation, say f, by successive generalizations of

shallow constructor terms until $f(X_1, \ldots, X_n)$ is obtained. A *shallow* constructor term is a term rooted by a constructor whose arguments are variables. For example, both `[]` and `[X|Xs]` are shallow constructor terms of the type list. The generalization replaces a shallow constructor term with a fresh variable. A plausible sequence originating from each rule of operation `take` defined in Example 8 is shown below. The sequences should be read from the bottom up, i.e., each term of a sequence, except the bottom one, is a generalization of the term below it. The bottom term is a variant of a rule's left-hand side.



Observe that modulo a renaming of variables, each sequence when traversed from top to bottom is a path from the root to a leaf of the definitional tree of the operation `take` shown in Example 8. The choice of which shallow constructor term to generalize is non-deterministic. Different choices from those presented above are possible for the first two (from the left) sequences, for example, `take (s N) []` could be generalized to `take M []`. This choice would prevent building a definitional tree of `take`.

In the second phase of the algorithm, the sequences computed in the first phase are assembled into a tree, if possible. This proceeds recursively as follows. Observe that all the sequences have the same head modulo a renaming of variables. The heads are merged to become the root of a (sub)tree. The tails of the sequences are partitioned, if possible, into subsets of sequences such that: (1) all the sequences in a subset have the same head modulo a renaming of variables; and (2) the heads of sequences in different subsets differ only for shallow constructor terms in the same position, which becomes an inductive position. For the example being discussed, this partition places the first two subsequences in one set and the remaining subsequence in another.



If such a partition is possible, the second phase is recursively applied to each subset; otherwise the computation of the definitional tree fails. In this example, the two occurrences of `take (s N) Y` will be merged and become the root of a subtree. If a defined operation has a definitional tree, the above algorithm computes it under the assumption that the non-deterministic choices of the first phase are angelic.

The needed narrowing strategy, as well as the other strategies based on definitional trees, take into account only the left-hand sides of the rules of a TRS, i.e., the right-hand sides do not play any role in the computation of a narrex. A consequence of this approach is that the definition of the strategy and its applicability is independent of extra variables, which may occur in the right-hand sides only. Therefore, the property of being inductively sequential is meaningful for TRSs with extra variables and needed narrowing steps can be computed for terms of these systems. For this reason, needed narrowing is applied, e.g., in Hanus et al. (2003), to inductively sequential TRSs with extra variables, but its soundness and completeness are proved (Antoy et al., 2000) for TRSs without extra variables.

3.2. Weakly orthogonal TRSs

The weakly orthogonal TRSs are a proper superclass of the inductively sequential TRSs. Rewrite rules in this class can overlap, but only if their corresponding critical pairs are trivial (syntactically equal). The rules's left-hand sides are patterns and consequently they can overlap only at the root. Therefore, weakly orthogonal constructor-based TRSs are almost orthogonal. Computations in this class are sometimes referred to as parallel, and so implemented. However, this class admits sequential normalizing rewrite strategies, as well, e.g., Kennaway (1989). There is no meaningful notion of needed redex for this class.

Example 10. An emblematic non-inductively sequential operation in this class, called *parallel-or*, is defined by the rules:

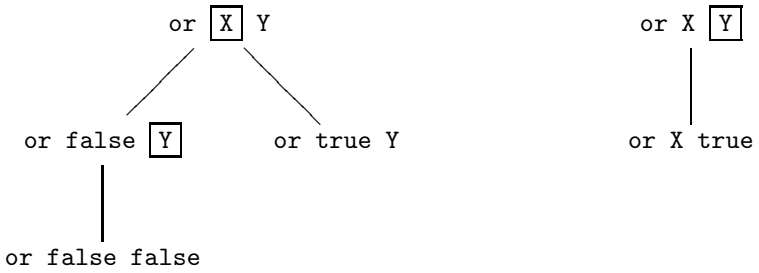
```
or true _ -> true                    R6
or _ true -> true                    R7
or false false -> false              R8
```

The term `or` (`or true` u) (`or` v `true`) has no needed redex regardless of the terms u and v.

Several practical normalizing strategies are known for computations in this class. The *parallel outermost* strategy (O'Donnell, 1977), as the name suggests, contracts simultaneously, or in parallel, the set of all the outermost redexes of a term. The *weakly needed* strategy (Sekar and Ramakrishnan, 1993) equally contracts in parallel all the redexes of a set called *necessary*. For a term t, a necessary set S of redexes of t is contained, possibly strictly, in the set of the outermost redexes of t and has the property that in any computation of t to a normal form a redex of S is eventually contracted. A necessary set of redexes is not difficult to compute. This strategy is optimal for arbitrary reductions (Sekar and Ramakrishnan, 1993) when the necessary sets of each term of a computation is minimal.

The weakly needed rewrite strategy can be formulated by means of definitional trees as well. It is easy to verify that the *parallel-or* operation has no definitional tree. Generalized forms of definitional trees for operations of this kind have been proposed, e.g., in Antoy (1991, 1992), Loogen et al. (1993). Informally, a generalized tree allows more than one inductive variable in a branch. A simpler viewpoint is to partition the rules of an operation into subsets for which there exists an ordinary definitional tree. Of course, this is trivial when each subset is a singleton, although a definitional tree may exist for larger subsets of rules. For the rules of Example 10, a possible partition is $\{R_6, R_8\} \uplus \{R_7\}$. A graphical

representation of the resulting trees is shown below.



Now, an operation may have several "smaller" or partial trees rather than a single one that includes all (the left-hand sides of) the rules. A redex of a term t may be computed as it would be for the inductively sequential TRSs, but if an operation is not inductively sequential, one of its partial trees is arbitrarily chosen. The union of the redexes computed over all the choices of possibly partial trees is a necessary set. This rewrite strategy, formalized in Antoy (1992), is equivalent to Sekar and Ramakrishnan (1993). Its extension to narrowing, known as *weakly needed narrowing* (Antoy et al., 1997), is therefore straightforward. However, the resulting narrowing strategy does not have the same characteristics of the corresponding rewrite strategy.

Example 11. Consider again the operation *parallel-or* defined in Example 10 and the term $t =$ `or U true`, where `U` is an uninstantiated variable. Weakly needed narrowing computes three steps, which are also complete evaluations, of t:

$$\texttt{or U true} \leadsto_{\{\texttt{U} \mapsto \texttt{true}\}} \texttt{true}$$
$$\texttt{or U true} \leadsto_{\{\texttt{U} \mapsto \texttt{false}\}} \texttt{true}$$
$$\texttt{or U true} \leadsto_{\{\}} \texttt{true}$$

It is clear a posteriori that the first two steps are subsumed by the third one.

In general, a strategy cannot easily determine if a computation is unnecessary without lookahead. A refinement of weakly needed narrowing, *parallel narrowing* (Antoy et al., 1997), avoids some unnecessary computations at the expenses of a substantial increase in complexity. Among the steps of a term computed by weakly needed narrowing, parallel narrowing discards, loosely speaking, certain steps with non-minimal substitutions, such as those in the previous example, and certain steps with non-outermost narrexes. The details are quite technical. It turns out that if a discarded step is necessary to ensure the completeness of a computation, an equivalent step will eventually be computed again later.

The decision of which steps to discard is based on an analysis of all the steps computed for a term. Therefore, there are situations in which parallel narrowing performs unnecessary computations because essential information becomes available in later steps. The following example from Antoy et al. (1997) shows this point.

Example 12. Extend the TRSs of Example 10 with the following definitions:

```
f 0 X -> X
h 0 -> true
```

and consider the term $t =$ `or (f U (h V)) (f V (h U))`. Parallel narrowing computes two derivations of t beginning with different unifiers eventually to discover that these derivations compute the same value and substitution. Therefore, one derivation is redundant. These two derivations are shown below:

$$t \leadsto_{\{U \mapsto 0\}} \texttt{or}\,(\texttt{h}\,\texttt{V})\,(\texttt{f}\,\texttt{V}\,\texttt{true}) \leadsto_{\{V \mapsto 0\}} \texttt{or}\,\texttt{true}\,\texttt{true} \leadsto_{\{\}} \texttt{true}$$
$$t \leadsto_{\{V \mapsto 0\}} \texttt{or}\,(\texttt{f}\,\texttt{U}\,\texttt{true})\,(\texttt{h}\,\texttt{U}) \leadsto_{\{U \mapsto 0\}} \texttt{or}\,\texttt{true}\,\texttt{true} \leadsto_{\{\}} \texttt{true}$$

3.3. Overlapping inductively sequential TRSs

The overlapping inductively sequential TRSs are a proper superclass of the inductively sequential TRSs. They are incomparable with the weakly orthogonal TRSs. As the name suggests, rewrite rules in overlapping inductively sequential TRSs can overlap, but it is also required that each operation is inductively sequential, i.e., it admits a definitional tree. This implies that two rules can overlap only if their left-hand sides are equal modulo a renaming of variables. By contrast to the case for weakly orthogonal TRSs, no restriction is placed on the right-hand sides of overlapping rewrite rules. By contrast to the case for inductively sequential and the weakly orthogonal TRSs, systems in this class are not confluent. For this reason, computations in this class are sometimes referred to as non-deterministic.

Example 13. The following operations define an alphabet (of digits) and the (non-empty) regular expressions parameterized by a given alphabet. In this context, the (meta)symbol "|" defines alternative right-hand sides of a same left-hand side:

```
digit -> "0" | "1" | ... | "9"
regexp X -> X
          | "(" ++ regexp X ++ ")"
          | regexp X ++ regexp X
          | regexp X ++ "*"
          | regexp X ++ "|" ++ regex X
```

The definition of operation `regexp` closely resembles the formal definition of *regular expression*. Non-deterministic operations contribute to the expressive power of a functional logic language. For example, to recognize whether a string, say $s =$ `"(01)*"`, denotes a well-formed regular expression over the alphabet of digits it simply suffices to evaluate (`regexp digit` $= s$). For parsing purposes, a less ambiguous definition that also accounts for the usual operator precedence would be preferable, but these aspects are irrelevant to the current discussion.

Non-deterministic operations not only contribute to the expressiveness of a program, they are also essential to preserve the inherent laziness of some computations. The following example (Antoy, 1997) shows this point.

Example 14. To keep the size of this example small, the problem is abstract. Suppose that `ok` is a unary function that, for all arguments, evaluates to `true` (although `ok` is contrived,

functions that do not look at all their arguments all the time are frequent) and `double` is a function of a natural number that doubles its argument:

```
ok _ -> true

double 0     -> 0
double (s X) -> s (s (double X))
```

Evaluating whether the "*double* of some expression *t* is *ok*", i.e., solving the goal:

```
ok (double t)
```

succeeds regardless of *t*, i.e., even if *t* is undefined.

Now, extend the program with a mechanism to halve numbers. I call it "mechanism", rather than function, because halving odd numbers rounds non-deterministically. Even numbers are halved as usual. Following the standard practice of logic programming, `half` is encoded by a predicate:

```
half 0         0     -> true
half (s 0)     0     -> true
half (s 0)     (s 0) -> true
half (s (s X)) (s Y) -> half X Y
```

Now, to find out whether the "*half* of some expression *t* is *ok*", one must solve the goal:

```
and (half t U) (ok U)
```

where `U` is a free variable and `and` is the usual boolean conjunction. Solving this goal requires to evaluate *t*. The computation is unnecessarily inefficient and it may even fail, if *t* cannot be evaluated to a natural. However, the analogous computation for `double` always succeeds.

The loss of laziness shown by the previous example is due to the fact that the producer of values, the expression rooted by `half`, is not functionally nested inside its consumer, the application of `ok`. Overlapping inductively sequential TRSs overcome this limitation. The following non-deterministic operation replaces the predicate `half` of the previous example:

```
half 0         -> 0
half (s 0)     -> 0 | s 0
half (s (s X)) -> s (half X)
```

and allows nesting the call to `half` inside `ok`:

```
ok (half t)
```

This goal is evaluated as the corresponding goal of `double`.

The evaluation strategy for overlapping inductively sequential TRSs is called *inductively sequential narrowing strategy* or *INS* (Antoy, 1997). This strategy has been formulated for narrowing computations since its inception, i.e., it does not originate from an earlier rewrite strategy. *INS* steps and needed narrowing steps are computed similarly.

An overlapping inductively sequential operation has a definitional tree exactly as a non-overlapping inductively sequential operation. The computation of an *INS* step goes as follows. First, one or more definitional trees are used to find a narrex in a term t, i.e., a position p of t and a unifier σ such that $\sigma(t|_p)$ is a redex. This is the same as for needed narrowing. However, the narrex computed by *INS* may have several replacements. Thus, *INS* non-deterministically computes a set of narrowing steps, whereas needed narrowing non-deterministically computes a single narrowing step. This difference entails significant differences in the properties of the two strategies.

By contrast to needed narrowing, *INS* is not hypernormalizing on ground terms. If a ground term is reducible to a value, then there exists no needed narrowing derivation that computes an infinite number of needed narrowing steps. Example 15 shows that this does not hold for *INS*. By contrast to needed narrowing, *INS* may execute avoidable steps. Every step of a needed narrowing computation to a value is unavoidable. Example 15 shows that this does not hold for *INS*.

Example 15. Consider the following non-deterministic defined operation:

```
f -> f | 0
```

The term `f` has a unique normal form, `0`. On `f`, *INS* computes, among others, the following derivation:

```
f -> f -> 0
```

where the initial step is clearly useless.

Despite these differences, *INS* shares, or better it extends, a crucial property of needed narrowing — and one of the most desirable properties of a strategy: every *INS* step is root-needed modulo a non-deterministic choice. A step s of a term t is *root-needed* (Middeldorp, 1997) if every derivation of t to a root-stable term eventually executes s. These steps are more fundamental than ordinary needed steps, since they allow the computation of infinitary normal forms. An *INS* step of a term t determines a narrex s and a substitution σ. The term $\sigma(t)$ cannot be evaluated to a constructor-rooted term unless the redex $\sigma(s)$ is contracted. This redex may have several replacements, but not all the replacements may be needed to compute a constructor-rooted term. *INS* computes all possible replacements and it seems unlikely that in general useless replacements can be determined without lookahead. The qualification "modulo a non-deterministic choice" in the formulation of the root-needed property is vacuous for needed narrowing since needed narrowing has no choices of replacements. If non-determinism is used appropriately, e.g., when the programmer has no information to make a choice, one can argue that *INS* does the best possible job under its conditions of employment.

Non-deterministic operations require rethinking some semantic aspects of both evaluation and strategies. For example, the meaning of the equality operation must be generalized, i.e., $t = u$ means that t and u have a common value (González Moreno et al., 1999) — one out of possibly many. Referring to Example 13, both equations `digit = "0"` and `digit = "1"` hold, but one should not infer that `"0" = "1"`. A more subtle issue is related to the events that should bind a *value* to a variable. The following example shows this point.

Example 16. The following function `min` is intended to compute the minimum of two natural numbers.

```
min a b -> if a<=b then a else b
```

The "less then or equal to" relational operator "`<=`" was defined in Example 6. The "**if** · **then** · **else** ·" construct can be defined, as a ternary function, by simple, ordinary rewrite rules which are irrelevant to this discussion.

Consider the term t = `min (half (s 0)) 0`, where `half` is the non-deterministic operation defined in Example 14. The term t is a redex. According to the rule of `min`, t is reduced to u = `if half (s 0)<=0 then half (s 0) else 0`. Now, if the first occurrence of `half (s 0)` is reduced to `0` and the second to `s 0`, the value of t is not what the programmer intended.

The intended behavior of the rule of `min` is to bind the same value to all the occurrences of the variable `a`. This behavior is referred to as *call-time* choice semantics (Hussmann, 1992) and it is an automatic consequence of eager or call-by-value strategies.

By contrast, the operation `regexp` discussed earlier has rules with two occurrences of variable `X`. This variable is bound in the initial application of `regexp` to a term, e.g., `digit`, which may eventually be reduced to a one-character string of a given alphabet. In this case, however, the intended meaning is opposite of that of the rule of the operation `min`. Unless all the occurrences of `X` bound to `digit` are evaluated independently of each other, some regular expressions would not be generated. In this case, the intended behavior is not to bind the same value to all the occurrences of `X`. This behavior is referred to as *need-time* choice semantics and it cannot be provided by eager or call-by-value strategies. The opportunity to compute with the need-time choice semantics tends to occur more frequently when a computation is parameterized by another non-deterministic computation.

In each case, the intended behavior seems to depend only on the program. The need-time choice semantics would be unsound for the operation `min` shown earlier. Likewise, the call-time choice semantics would be incomplete for the operation `regexp` shown earlier. When non-deterministic computations model the behavior of a program, as it is the case of functional logic computations, the programming language should allow the programmer to encode in the program the intended semantics.

Functional logic languages such as Curry and $\mathcal{TOY}$ adopt the call-time choice semantics only. Unrestricted rewriting and narrowing provide the need-time choice semantics which would be unintended for these languages. Nevertheless, the operational behavior of these languages is defined via definitional trees, e.g., see González Moreno et al. (1999), Hanus et al. (2003), and hence via the strategies discussed in this paper, to achieve a more satisfactory performance. A simple implementation technique, namely sharing the representation of all the occurrences of a variable, ensures the call-time choice semantics (Albert et al., 2002; Hussmann, 1992; Tolmach and Antoy, 2003). Compilers that map source functional logic code into Prolog code, e.g., Hanus et al. (2003), provide sharing easily with the technique described in Antoy and Hanus (2000).

The proofs of the soundness, completeness and optimality of *INS* are only sketched in Antoy (1997).

3.4. Constructor-based TRSs

The constructor-based TRSs are the largest class proposed to model functional logic programs. This class is a proper superclass of all the other classes discussed previously. Overlapping of the rules' left-hand sides is unrestricted, though in constructor-based TRSs the left-hand sides of two rules can overlap only at the root. No specific restrictions are imposed on the right-hand sides of overlapping rules.

Example 17. The following rewrite rules define an operation, `permute`, that non-deterministically computes a permutation of a list. The operation `insert` does not belong to any of the previously discussed classes of TRSs:

```
permute [] -> []
permute [X|Xs] -> insert X (permute Xs)
insert X Ys -> [X|Ys]
insert X [Y|Ys] -> [Y|insert X Ys]
```

A potential drawback of computing with a class as large as the constructor-based TRSs is that outermost rewrite strategies are not normalizing; hence outermost narrowing strategies are not complete. All the strategies discussed in the previous sections are outermost, a condition that simplifies reasoning about computations and consequently helps proving properties such as completeness and optimality. Referring to the previous example, consider the evaluation of the term $t = \texttt{insert}\ u\ v$. This term is a redex regardless of the values of u and v. However, one may have to evaluate v to apply the second rewrite rule of `insert`. If this rule is never applied, some permutations of some lists cannot be computed (Antoy, 2001).

Efficient strategies for the whole class of the constructor-based TRSs are not as well developed as those for the other classes discussed in this paper. An early rewrite strategy for this class is proposed in Antoy (1991). A generalization to narrowing, restricted to the weakly orthogonal TRSs is proposed in Loogen et al. (1993). Both strategies are based on some form of generalized definitional tree. The completeness of these strategies is unknown. These strategies are commonly referred to as "demand driven", which informally means the following. A subterm v of a term t is evaluated if there is a rule R potentially applicable to t that demands the evaluation of v, i.e., R cannot be applied if v is not further evaluated. However, the application of R to t may not be necessary for the whole computation in which t occurs. Demand-driven strategies inspire confidence in their completeness since they try to create the conditions for the application of every possible rewrite rule to a term. They are also wasteful if the application of a rule to a term and consequently the evaluation of subterms for the application of that rule are unnecessary.

The lack of well-defined strategies with provable properties motivated alternative efforts for computations in this class, e.g., a narrowing calculus (González Moreno et al., 1999), a program transformation (Antoy, 2001), and a compilation based on an abstract interpretation (Marino and Moreno-Navarro, 2000). The latter is beyond the scope of this paper. The narrowing calculus, CRWL, is mainly proposed as the semantics of narrowing computations, but it is not well suited as a practical evaluation mechanism.

The implementation of narrowing computations in $\mathcal{TOY}$, which is modeled by CRWL, is based on definitional trees. Narrowing calculi will be briefly addressed later.

The program transformation (Antoy, 2001), called *sequentialization*, maps a computation of a constructor-based, possibly conditional, TRS $\mathcal{R}$ into a computation of an overlapping inductively sequential TRS $\mathcal{R}'$. Computations in $\mathcal{R}'$ are then executed by *INS*. The transformation extends the signature of $\mathcal{R}$ with new operation symbols, but no new constructors. The change in signature generally creates new steps and new normal forms. However, the transformation is intended to preserve the computations of $\mathcal{R}$ in the sense that any term built over the signature of $\mathcal{R}$ is evaluated to the same set of values by the rules of both $\mathcal{R}$ and $\mathcal{R}'$. The additional normal forms of $\mathcal{R}'$ contain defined operations and consequently represent failed computations. Computations executed by *INS* are optimal with respect to the rewrite rules of $\mathcal{R}'$, but not necessarily with respect to the rewrite rules of $\mathcal{R}$. The correctness of the transformation has not been formally proved yet.

The sequentialization transformation is relatively straightforward. Every inductively sequential, possibly overlapping, operation of $\mathcal{R}$ is unchanged in $\mathcal{R}'$. If f is not an overlapping inductively sequential operation of $\mathcal{R}$, $\mathcal{R}'$ introduces a new function f_i for every rule $l_i \to r_i$ of f in $\mathcal{R}$ and redefines f as follows:

$$f(X_1, \ldots, X_n) \to f_1(X_1, \ldots, X_n) \mid \ldots \mid f_k(X_1, \ldots, X_n)$$

where $X_1, \ldots, X_n$ are variables and k is the number of rules defining f in $\mathcal{R}$. Furthermore, f_i is defined by the single rule:

$$l_i' \to r_i$$

where l_i' is equal to l_i except that its root symbol is f_i instead of f.

In short, an overlapping of two rules' left-hand sides is transformed into a choice of right-hand sides. The following example shows how this works in practice.

Example 18. The following rules are the sequentialization of the operation `insert` defined in Example 17, which is not inductively sequential:

```
insert X Y -> insert1 X Y | insert2 X Y
insert1 X Y -> [X | Y]
insert2 X [Y | Ys] -> [Y | insert X Ys]
```

It is easy to verify that, for any terms u and v, the values of (`insert` u v) computed using the operation `insert` defined in Example 17 are the same as those computed by its sequentialization.

Several refinements and optimizations are applicable to the above scheme. For example, the operation $\texttt{insert}_1$ can be entirely eliminated by unfolding. In particular, a better operational behavior is obtained when it is possible to apply the transformation, to proper subsets of the set of all the rules defining a function (López-Fraguas and Sánchez-Hernández, 2001) rather than to the entire set. Observe that the sequentialization of `insert` creates some normal forms that do not exist in the original TRS, e.g., $\texttt{insert}_2\,0\,[]$. Since this term contains the defined operation $\texttt{insert}_2$, it does not represent a (successful) computation.

4. Related issues

The previous sections have glossed over some important issues related to functional logic computations. These issues are addressed in this section. The focus, as in the rest of this paper, is on evaluation strategies.

4.1. Left-linearity

All the classes of TRSs discussed earlier were assumed to be left-linear. The inductively sequential TRSs, whether or not overlapping, and the weakly orthogonal TRSs are left-linear by definition, but the larger class of the constructor-based TRSs was intentionally limited. To understand the issues behind left-linearity consider the operation `f` defined by the rule:

```
f(X,X) -> r
```
(2)

The problem is to determine when a term `f(`u`,`v`)`, for some terms u and v, should be contracted with the above rule. Obviously, it should be contracted if u and v are identical. However, this leaves out a term such as `f(2+2,4)`, which seems inappropriate for programming. Therefore, a better approach is to contract `f(`u`,`v`)`, if u and v are "equal". Thus, the problem becomes to define a suitable notion of equality in this case. One approach would be to consider u and v equal if they can be narrowed to unifiable terms, but this cannot be easily determined.

Example 19. Consider the following program:

```
from N -> [N | from (N+1)]
tailstar [_ | T] -> tailstar T
```

and the terms $s =$ `tailstar(from 2)` and $t =$ `tailstar(from 0)`. These terms are joinable, i.e., there exists a term u such that $s \overset{*}{\rightarrow} u \overset{*}{\leftarrow} t$. However, despite the fact that the above program is well behaved, e.g., it is inductively sequential, there are no reasonably efficient general strategies for determining the joinability of s and t.

Therefore, joinability does not seem an appropriate choice of equality, at least for programming. Strict equality seems more reasonable since it can be defined by ordinary rewrite rules and it can be tested with the strategies discussed earlier. Therefore, it is the choice of functional logic languages such as Curry and $\mathcal{TOY}$.

With this notion of equality, the terms s and t of the previous example are not considered equal, although they are joinable. On the positive side, left-linearity is no longer a restriction because the meaning of (2) is the same as the following left-linear conditional rule:

```
f(X,Y) -> r :- X = Y
```
(3)

which can be freely coded in a program.

4.2. Conditional rules

The classes of TRSs discussed earlier are all unconditional. The outermost-fair rewrite strategy, which is normalizing for almost orthogonal TRSs (O'Donnell, 1977), is also normalizing for conditional almost orthogonal TRSs (Bergstra and Klop, 1986). The strategies discussed in Section 3 are based, either directly or indirectly, on definitional trees. Definitional trees depend on the left-hand sides of rewrite rules only. Therefore, strategies defined through definitional trees are somewhat independent of whether TRSs are conditional. An approach that takes advantage of this consideration is based on a program transformation, called *deconditionalization* (Antoy, 2001).

This transformation maps a computation in a conditional constructor-based TRS $\mathcal{R}$ into an equivalent computation in an unconditional constructor-based TRS $\mathcal{R}'$. Computations in $\mathcal{R}'$ are executed as discussed in Section 3.4. The transformation extends the signature of $\mathcal{R}$ with a single new operation symbol, but no new constructors. The change in signature generally creates new steps and new normal forms. However, the transformation is intended to preserve the computations of $\mathcal{R}$ in the sense that any term built over the signature of $\mathcal{R}$ is evaluated to the same set of values by the rules of both $\mathcal{R}$ and $\mathcal{R}'$. The additional normal forms of $\mathcal{R}'$ contain defined operations and consequently represent failed computations. The correctness of the transformation has not been formally proved yet.

Similarly to the sequentialization transformation presented earlier, the deconditionalization transformation is relatively straightforward. In short, the condition of a conditional rewrite rule is moved to the right-hand side using the newly introduced operation. More precisely, a conditional rewrite rule of the form:

$$l \texttt{ -> } r \texttt{ :- } t_1 \texttt{ = } u_1\texttt{, } \ldots\texttt{, } t_n \texttt{ = } u_n$$

is transformed into:

$$l \texttt{ -> if } t_1 \texttt{ = } u_1 \texttt{ \& } \cdots \texttt{ \& } t_n \texttt{ = } u_n \texttt{ then } r$$

where, as expected, the "**if** $\cdot$ **then** $\cdot$" binary operation returns its second argument when its first argument succeeds, and `&` is the constrained conjunction operator defined earlier.

4.3. Narrowing calculi

Narrowing *calculi* have been investigated as alternatives to narrowing *strategies*, e.g., LCN (Middeldorp et al., 1996) for confluent TRSs, OINC (Ida and Nakahara, 1997) for orthogonal TRSs and goals whose right-hand side is a ground normal form, CLNC (González Moreno et al., 1999) for left-linear constructor-based TRSs. A common reason advocated for studying a calculus rather than a strategy is that "narrowing is a complicated operation" (Middeldorp and Okui, 1998). Calculi also shed light and help formalizing other issues of computations such as non-determinism, sharing and the behavior of strict equality.

Calculi come in various flavors, but generally they consist of a handful of *inference* or *transformation* rules for equational goals. Calculi ease the proofs of soundness and completeness by simulating narrowing steps by means of a small number of more elementary inference rules. This fragmentation sometimes increases the non-determinism

of a computation and makes implementations less efficient. Some calculi have been refined to alleviate this problem, e.g., LCN_d (Middeldorp and Okui, 1998) for left-linear confluent constructor-based TRSs with strict equality, and S-OINC (Ida and Nakahara, 1997).

Strong optimality properties have been claimed for narrowing strategies more often than for narrowing calculi. The implementation of narrowing is still the subject of active investigation, but it seems safe to guess that, in general, strategies can be implemented more easily and efficiently than calculi because strategies are more directly related to the terms that are the object of a computation.

4.4. Higher-order functions

The final relevant issue about functional logic programming neglected earlier concerns high-order computations, a cornerstone of functional programming. Higher-order functions, i.e., functions that take other functions as arguments, contribute to the expressive power of a language by parameterizing computations over other computations. A typical example is the function `map`, which applies some function to all the elements of list:

```
map _ [] -> []
map F [X | Xs] -> [F X | map F Xs]
```

The difference with respect to previous examples is that the first argument of `map` does not evaluate to a value, but to an operation.

The theory of higher-order rewriting is not as advanced as that of (first-order) rewriting, thus not as much is known about rewrite strategies for higher-order TRSs. However, the well-known outermost-fair rewrite strategy, which is normalizing for almost orthogonal TRSs (O'Donnell, 1977), is normalizing also for weakly orthogonal higher-order TRSs if an additional condition, full extension, is imposed on higher-order rewrite rules (van Raamsdonk, 1999). The theory of higher-order narrowing is even less developed. Similar to the first-order case, several classes of higher-order TRSs have been proposed for higher-order functional logic computations, e.g., *SFL* programs (González-Moreno, 1993; González-Moreno et al., 1997, 2001), *applicative* TRSs (Nakahara et al., 1995), and *higher-order* inductively sequential TRSs (Hanus and Prehofer, 1996). Different approaches have been adopted to prove properties of functional logic computations in these classes. Computations in SFL programs are mapped to first-order computations by a transformation that extends to narrowing a well-known transformation for higher-order logic computations (Warren, 1982). Computations in applicative TRSs are executed by a calculus that makes inference steps of a granularity finer than narrowing steps. Computations in higher-order inductively sequential TRSs are executed using another generalization of definitional trees.

A significant difference between functional logic computations and functional computations is that narrowing is capable of synthesizing functions. In many cases, functions of this kind would be the result of a top-level computation. For example, solving for `X` the constraint:

```
map X [0,1,2] = [2,3,4]
```

would return, among other possibilities, the computed answer $\{\mathtt{X} \mapsto \mathtt{s} \circ \mathtt{s}\}$, where $\mathtt{s}$ is the constructor defined in Example 8 and "$\circ$" is the functional composition operator typically available in functional and functional logic languages. This kind of higher-order results can be computed also by means that do not involve narrowing, e.g., Nadathur and Miller (1988). Most implementations of functional languages are not equipped to deal with this possibility. When the result of a computation is a function, functional languages report so, but do not identify in any expressive form which function. This design choice would seem to indicate that higher-order results are not particularly interesting, at least in functional programming. Narrowing considerably expands the power of functional evaluations, and for this reason higher-order narrowing is being investigated, too, e.g., see Anastasiadis and Kuchen (1996); Prehofer (1994), but the feasibility and usefulness of computing higher-order results has not yet been clearly established.

As in other situations, and for the same reasons, transformational approaches have been proposed for higher-order computations as well. In short, terms with partially applied symbols are transformed into terms built with new symbols introduced for this purpose. Every symbol in a transformed term is fully applied. The original idea (Warren, 1982) is formulated for functional evaluation in logic programming, (González-Moreno, 1993) generalizes it to narrowing, and (Antoy and Tolmach, 1999) refines it by preserving type information which may dramatically reduce the size of the narrowing space. These approaches are interesting because they extend non-trivial results proved for first-order strategies to the higher-order case with a modest conceptual effort.

5. Conclusion

This paper offers an overview of evaluation strategies for functional logic programs. A program is seen as a constructor-based TRS and an evaluation or computation is a narrowing derivation to a value — a constructor normal form. Constructor-based TRSs are good models for programs because they compute with functions defined over algebraic data types. Non-constructor-based TRSs are seldom used as programs.

I presented four subclasses of the constructor-based TRSs. Each subclass captures some interesting aspect of computing, such as parallelism or non-determinism. Computations in different classes are best accomplished by different strategies. For each class, I presented a narrowing strategy and, in some cases, the rewrite strategy from which it originates. For smaller classes, the strategies are better understood, i.e., rewrite strategies are normalizing, they compute the value, if it exists, of a term, and narrowing strategies are sound and complete, i.e., when used to solve an equation they compute only and all the equation's solutions. For larger classes, the properties of the strategies have been proved to varying degrees of rigor, but these are the strategies most interesting in practice and adopted in programming languages' implementations. Not surprisingly, as classes get bigger the claims about the efficiency of strategies suitable for these classes get weaker.

Finally, I considered two extensions of the constructor-based TRSs which are important for programming: conditional and higher-order rewrite rules. Strategies for functional logic computations in these extensions are not as well developed as the ordinary case. Transformations from extended TRSs to ordinary TRSs make it possible to reuse the

strategies presented earlier and take advantage of the intellectual efforts invested in their development.

Acknowledgements

I would like to thank Bernhard Gramlich and Salvador Lucas Alba for inviting me to write a preliminary version of this paper for the International Workshop on Reduction Strategies in Rewriting and Programming held in Utrecht, The Netherlands, in May 2001.

The anonymous reviewers made countless useful suggestions which have been incorporated in the final version. The author was supported in part by the NSF grants CCR-0110496 and CCR-0218224.

References

Aït-Kaci, H., 1990. An overview of life. In: Schmidt, J., Stogny, A. (Eds.), Proc. Workshop on Next Generation Information System Technology. In: LNCS, vol. 504. Springer, pp. 42–58.

Albert, E., Hanus, M., Huch, F., Oliver, J., Vidal, G., 2002. Operational semantics for functional logic languages. In: Comini, M., Falaschi, M. (Eds.), Electronic Notes in Theoretical Computer Science, vol. 76. Elsevier Science Publishers, Available at http://www.elsevier.nl/locate/entcs/volume76.html.

Anastasiadis, J., Kuchen, H., 1996. Higher order Babel: Language and implementation. In: Dyckhoff, R., Herre, H., Schroeder-Heister, P. (Eds.), Proc. of the 5th International Workshop on Extensions of Logic Programming. ELP'96, March. In: LNCS, vol. 1050. Springer, Leipzig, Germany.

Antoy, S., 1991. Non-determinism and lazy evaluation in logic programming. In: Clement, T.P., Lau, K.-K. (Eds.), Logic Programming Synthesis and Transformation. LOPSTR'91, July. Springer-Verlag, Manchester, UK, pp. 318–331.

Antoy, S., 1992. Definitional trees. In: Proc. of the 4th Intl. Conf. on Algebraic and Logic Programming. In: LNCS, vol. 632. Springer, pp. 143–157.

Antoy, S., 1997. Optimal non-deterministic functional logic computations. In: Proc. of the 6th International Conference on Algebraic and Logic Programming. ALP'97. In: LNCS, vol. 1298. Springer, pp. 16–30.

Antoy, S., 2001. Constructor-based conditional narrowing. In: Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming. PPDP'01, September, Florence, Italy. pp. 199–205.

Antoy, S., Echahed, R., Hanus, M., 1997. Parallel evaluation strategies for functional logic languages. In: Proc. of the 14th International Conference on Logic Programming. ICLP'97. MIT Press, pp. 138–152.

Antoy, S., Echahed, R., Hanus, M., 2000. A needed narrowing strategy. Journal of the ACM 47 (4), 776–822.

Antoy, S., Hanus, M., 2000. Compiling multi-paradigm declarative programs into Prolog. In: Proc. of the 3rd International Workshop on Frontiers of Combining Systems. FroCoS 2000, March. In: LNCS, vol. 1794. Springer, Nancy, France, pp. 171–185.

Antoy, S., Tolmach, A., 1999. Typed higher-order narrowing without higher-order strategies. In: 4th Fuji International Symposium on Functional and Logic Programming. FLOPS'99. In: LNCS, vol. 1722. Springer, Tsukuba, Japan, pp. 335–350.

Baader, F., Nipkow, T., 1998. Term Rewriting and All That. Cambridge University Press.

Barry, B., 1996. Needed narrowing as the computational strategy of evaluable functions in an extension of Gödel. Master's Thesis, Portland State University.

Bergstra, J.A., Klop, J.W., 1986. Conditional rewrite rules: Confluence and termination. Journal of Computer and System Sciences 32 (3), 323–362.

Bezem, M., Klop, J.W., de Vrijer, R. (Eds.), 2003. Term Rewriting Systems. Cambridge University Press.

Bosco, P., Giovanetti, E., Moiso, C., 1988. Narrowing vs. sld-resolution. Theoretical Computer Science 59 (1–2), 3–23.

Boudol, G., 1985. Computational semantics of term rewriting systems. In: Nivat, M., Reynolds, J.C. (Eds.), Algebraic Methods in Semantics. Cambridge University Press, Cambridge, UK (Chapter 5).

Brand, M.v.d., Klint, P., 2003. ASF + SDF Meta-Environment User Manual Revision : 1:134. Centrum voor Wiskunde en Informatica (CWI), 18 September, The Netherlands.

Dershowitz, N., 1995. 33 examples of termination. In: Comon, H., Jouannaud, J.-P. (Eds.), French Spring School of Theoretical Computer Science Advanced Course on Term Rewriting. May 1993, Font Romeux, France. vol. 909. Springer-Verlag, Berlin, pp. 16–26.

Dershowitz, N., Jouannaud, J., 1990. Rewrite systems. In: van Leeuwen, J. (Ed.), Handbook of Theoretical Computer Science B: Formal Methods and Semantics. North Holland, Amsterdam, pp. 243–320 (Chapter 6).

Dijkstra, E.W., 1976. A Discipline of Programming. Prentice-Hall.

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

González-Moreno, J.C., 1993. A correctness proof for Warren's HO into FO translation. In: Proc. GULP'93. October, Gizzeria Lido, Italy. pp. 569–585.

González-Moreno, J.C., Fraguas, F.J.L., González, M.T.H., Artalejo, M.R., 1999. An approach to declarative programming based on a rewriting logic. The Journal of Logic Programming 40, 47–87.

González-Moreno, J.C., González, M.T.H., Artalejo, M.R., 1997. A higher order rewriting logic for functional logic programming. In: Proc. of the 14th International Conference on Logic Programming. ICLP'97, July, Leuven, Belgium. pp. 153–167.

González-Moreno, J.C., González, M.T.H., Artalejo, M.R., 2001. Polymorphic types in functional logic programming. Journal of Functional and Logic Programming 2001 (1).

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

Hanus, M., 2003. Curry: An integrated functional logic language (vers. 0.8). Available at http://www.informatik.uni-kiel.de/~curry.

Hanus, M., Antoy, S., Engelke, M., Höppner, K., Koj, J., Niederau, P., Sadre, R., Steiner, F., 2003. PAKCS: The Portland Aachen Kiel Curry System. Available at http://www.informatik.uni-kiel.de/~pakcs.

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

Hanus, M., Prehofer, C., 1996. Higher-order narrowing with definitional trees. In: Proc. 7th International Conference on Rewriting Techniques and Applications. RTA'96. In: LNCS, vol. 1103. Springer, pp. 138–152.

Huet, G., Lévy, J.-J., 1991. Computations in orthogonal term rewriting systems. In: Lassez, J.-L., Plotkin, G. (Eds.), Computational Logic: Essays in Honour of Alan Robinson. MIT Press, Cambridge, MA, pp. 395–443.

Hussmann, H., 1992. Nondeterministic algebraic specifications and nonconfluent rewriting. Journal of Logic Programming 12, 237–255.

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

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

Klop, J.W., 1992. Term rewriting systems. In: Abramsky, S., Gabbay, D., Maibaum, T. (Eds.), Handbook of Logic in Computer Science, vol. II. Oxford University Press, pp. 1–112.

Lloyd, J., 1999. Programming in an integrated functional and logic language. Journal of Functional and Logic Programming (3), 1–49.

Loogen, R., Fraguas, F.L., Artalejo, M.R., 1993. A demand driven computation strategy for lazy narrowing. In: Proc. 5th International Symposium on Programming Language Implementation and Logic Programming. PLILP'93. In: LNCS, vol. 714. Springer, pp. 184–200.

López-Fraguas, F.J., Sánchez-Hernández, J., 1999. TOY: A multiparadigm declarative system. In: Proc. RTA'99. In: LNCS, vol. 1631. Springer, pp. 244–247.

López-Fraguas, F.J., Sánchez-Hernández, J., 2001. Functional logic programming with failure: A set-oriented view. In: Nieuwenhuis, R., Voronkov, A. (Eds.), Proc. 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. In: Lecture Notes in Computer Science, vol. 2250. Springer, pp. 455–469.

Marino, J., Moreno-Navarro, J., 2000. Using static analysis to compile non-sequential functional logic programs. In: Pontelli, E., Costa, V.S. (Eds.), Practical Aspects of Declarative Languages, Second International Workshop. PADL 2000, Proceedings, January 2000, Boston, MA, USA. In: Lecture Notes in Computer Science, vol. 1753. Springer, pp. 63–80.

Middeldorp, A., 1997. Call by need computations to root-stable form. In: Proc. 24th ACM Symposium on Principles of Programming Languages. Paris. pp. 94–105.

Middeldorp, A., Okui, S., 1998. A deterministic lazy narrowing calculus. Journal of Symbolic Computation 25 (6), 733–757.

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

Moreno-Navarro, J.J., Rodríguez-Artalejo, M., 1992. Logic programming with functions and predicates: The language BABEL. Journal of Logic Programming 12, 191–223.

Nadathur, G., Miller, D., 1988. An overview of λprolog. In: Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming. Seattle. MIT Press, pp. 810–827.

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

O'Donnell, M.J., 1977. Computing in Systems Described by Equations. In: LNCS, vol. 58. Springer.

O'Keefe, R.A., 1990. The Craft of Prolog. The MIT Press, Cambridge, MA.

Petersson, K., Smith, J.M., 1986. Program derivation in type theory: A partitioning problem. Computer Languages 11 (3–4), 161–172.

Peyton Jones, S.L., Hughes, J., 1999. Haskell 98: A non-strict, purely functional language. http://www.haskell.org.

Prehofer, C., 1994. Higher-order narrowing. In: Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science. IEEE Computer Society Press, Paris, France, pp. 507–516.

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

Tolmach, A., Antoy, S., 2003. A monadic semantics for core curry. In: Vidal, G. (Ed.), Electronic Notes in Theoretical Computer Science, vol. 86. Elsevier. Available at http://www.elsevier.nl/locate/entcs/volume86.html.

van Raamsdonk, F., 1999. Higher-order rewriting. In: Proceedings of the 10th International Conference on Rewriting Techniques and Applications. RTA '99. In: LNCS, vol. 1631. Springer, pp. 220–239.

Warren, D., 1982. Higher-order extensions to PROLOG: Are they needed? In: Machine Intelligence, vol. 10. pp. 441–454.