Evaluation Strategies for Functional Logic Programming

Sergio Antoy*

Department of Computer Science, Portland State University,
P.O. Box 751, Portland, OR 97207, U.S.A.
{antoy}@cs.pdx.edu

Abstract. Recent advances in the foundations and the development 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-confluents subclasses have been developed just for functional logic computations. In this paper, I will discuss the elements that play a relevant role in evaluation strategies for functional logic programming, describe some important classes of rewrite systems that model functional logic programs, show examples of the differences in expressiveness provided by these classes, and review the characteristics of narrowing strategies proposed for each class of rewrite systems.

1 Introduction

Functional logic programming studies programming languages that join in a single paradigm the features of functional programming and logic programming. For the most part, a functional logic program can be seen as a constructor-based conditional rewrite system (TRS). In the examples, I take several liberties with the notation.

Example 1. The following program solves the well-known *N-queens* problem:

```
queens X -> Y :- Y=permute X, void(capture Y)
permute [] -> []
permute [X|Xs] -> U++[X]++V :- U++V=permute Xs
capture Y :- _++[Y1]++K++[Y2]++_=Y, abs(Y1-Y2)=length K+1
```

* The author has been partially supported by the NSF grant INT-9981317.

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 symbol "$=$" is interpreted as strict equality. The program adopts the familiar Prolog notation for lists and variables and uses common infix arithmetic operators, but this is only syntactic sugar. `Abs`, which stands for integer absolute value, and `++` and `length`, which stand for list concatenation and length, are assumed to be library functions. `Capture` is a *constraint*, i.e., a function that, similar to a Prolog predicate, "succeeds" iff its condition succeeds. `Void` is a primitive construct that succeeds iff its argument is a constraint with no solution. A query of the form, e.g., `queens [1,2,3,4]` non-deterministically computes a solution of the *4-queens* problem.

To understand the promise of functional logic programming languages, it is instructive to compare the above program with textbook examples of both functional and logic programs proposed for the same problem. All the programs, including ours, are structured as a generate-and-test pattern. The generator generates a permutation of the rows of the chess-board. The n-th element of the permutation represents the placement on the chess-board of the queen in column n. An efficient solution should avoid, e.g., the complete generation of $(n-1)!$ permutations that start with an incorrect initial placement when a single test would suffice.

The pure logic version [21, pages 132–135] is complicated by the need to generate potential solutions incrementally and test them before generating the next increment. This prevents the use of library predicates, e.g., to compute permutations, and makes the code of this program specific to this problem and non-reusable. The pure functional version [10, pages 161–165] is complicated both by the presence of data structures, such as a list of lists that (lazily) holds the entire set of permutations, or a list of pairs that eases the test of the safety of a placement, and by the presence of functions that construct and take apart these structures.

The functional logic version is textually shorter and conceptually simpler. For example, generator and tester are functionally nested and lazily executed and there are no bookkeeping and control data structures. Key factors that contribute to this simplicity and are unavailable in either the functional or the logic program are: (1) *non-determinism*, e.g., operation `permute` computes one of the many permutations of its argument, (2) *semantic unification*, e.g., the variables in the constraint (`U++V=permute` $\mathtt{X_s}$) are instantiated, if possible, to satisfy the equation, and (3) *functional inversion*, i.e., the possibility to compute a value for some argument(s) of a function from a result, e.g., the expression (`_++[`$\mathtt{Y_1}$`]++K++[`$\mathtt{Y_2}$`]++_`) is used to extract, lazily and non-deterministically, sublists from a list rather than to concatenate them.

The increased expressive power of functional logic programs pose heavier demands on their execution. These demands involve two specific aspects of 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 non-deterministic operations, such as `permute`, and extra variables, such as `U`, `V` and `K`. In this paper, I discuss some classes of TRSs proposed for functional logic programming and suitable evaluation strategies for these classes. 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 [8, 11, 17] and functional logic programming [13].

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*. 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. $\textsc{Term}(\Sigma \cup \mathcal{X})$ is the set of *terms* constructed over Σ and a countably infinite set $\mathcal{X}$ of *variables*. $\textsc{Term}(\mathcal{C} \cup \mathcal{X})$ 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.

A *pattern* is a term of the form $f(t_1, \ldots, t_n)$, $n \geqslant 0$, where $f \in \mathcal{F}$ 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. Traditionally, it is required that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. This condition is not imposed here since it appears unnecessarily restrictive for functional logic computations. An unconditional TRS, $\mathcal{R}$, defines a rewrite relation $\to_{\mathcal{R}}$ on terms as follows: $s \to_{p,R} t$ if there exists a position p in s, a rewrite rule $R = l \to r$ with fresh variables and a substitution σ with $s|_p = \sigma(l)$ and $t = s[\sigma(r)]_p$. The instantiated left-hand side $\sigma(l)$ of a rewrite rule $l \to r$ 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 is of the form $l \rightarrow r :- c$, where l and r are defined as in the unconditional case and c is a *sequence of elementary equational constraints*, i.e., pairs of terms of the form $t = u$. The definition of the rewrite relation for conditional TRSs is fairly more complicated than for unconditional TRSs. The classic approach to conditional rewriting is discussed in [9].

A left-linear, conditional, constructor-based TRS is a good model for a functional or a logic program. Computations are (expressed by) operation-rooted terms ultimately applied to values.

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

```
data bool   = true | false
data list a = []   | [a | list a]
```

and operations are defined by rewrite rules such as those of Example 1. Identifiers `true` and `false` are the familiar Boolean constants. `[]` (empty list) and `[·|·]` (non-empty list) are the constructors of the polymorphic type lists. Identifier `a` is a type variable ranging over all types. A value or *data term* is a well-formed expression containing variables, constants and data constructors, e.g., `[x,y]` which stands for `[x|[y|[]]]`.

The fundamental computation of functional logic languages is *narrowing*. A term s *narrows* to t with substitution σ, denoted $s \rightsquigarrow_\sigma t$, if σ is an idempotent constructor substitution such that $\sigma(s) \rightarrow t$. A term s such that $\sigma(s)$ is a redex is called a *narrex* (*narr*owable *ex*pression). 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 can be suboptimal [6]. A *computation* or *evaluation* of a term s is a narrowing derivation $s = t_0 \rightsquigarrow_{\sigma_1} \ldots \rightsquigarrow_{\sigma_n} t_n = t$, where t is a value. 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 the type list. The infix operation "`&`" is the constraint conjunction. Identifier `success` denotes a solved constraint. It is explicitly represented in this paper to define computations using only rewrite rules, but with an appropriate syntax it could be eliminated from programs. In practice, strict equality would be a built-in operation of a functional logic language run-time system.

`[] ++ X -> X`	R_1
`[X|Y] ++ Z -> [X	Y++Z]`	R_2
`[] = [] -> success`	R_3
`[X|Xs] = [Y|Ys] -> X=Y & Xs=Ys`	R_4
`success & X -> X`	R_5

The execution of the program of Example 1 requires the solution of constraints, such as `U++V=[2,3,4]`, 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 instantiates variables `U` and `V`. Both the rule and the substitution applied in a step are shown to the right of the reduct:

`U++V=[2,3,4]`	
$\rightsquigarrow$ `[U1|Us++V]=[2,3,4]`	$R_2, \{U \mapsto [U_1 \vert U_s]\}$
$\rightsquigarrow$ `U1=2 & Us++V=[3,4]`	$R_4, \{\}$
$\rightsquigarrow$ `success & Us++V=[3,4]`	$R_i, \{U_1 \mapsto 2\}$
$\rightsquigarrow$ `Us++V=[3,4]`	$R_5, \{\}$
$\vdots$	

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 type integer. This solution instantiates variable `U` to a list with head `2`.

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 $t \rightsquigarrow_{p,l\to r,\sigma} s$ iff $s = \sigma(t[r]_p)$. Formally, a narrowing strategy is a mapping that takes a term t and yields a set of triples of the form $\langle p, l \to r, \sigma \rangle$ interpreted as narrowing steps as defined earlier.

Example 4. Continuing Example 3, a good narrowing strategy applied to the constraint `U++V=[2,3,4]` computes the following two steps: $\langle 1, R_1, \{U \mapsto []\}\rangle$ and $\langle 1, R_2, \{U \mapsto [U_1|U_s]\}\rangle$. The first step yields a solution with answer `U=[]` and `V=[2,3,4]`. The second step was shown earlier.

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 $t \overset{*}{\rightsquigarrow}_\sigma u$ implies $\sigma(t) \overset{*}{\to} u$. A strategy is *complete* iff $\sigma(t) \overset{*}{\to} u$ implies the existence a substitution $\eta \leqslant \sigma$ such that $t \overset{*}{\rightsquigarrow}_\eta u'$ with $u' \leqslant u$. Intuitively, both the soundness and the completeness of a strategy are best understood when the initial term of a derivation is an equational constraint containing occurrences of free 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. Two factors affect the efficiency of a strategy: (1) unnecessary steps should not be computed, and (2) steps should be computed without 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 naive way to ensure completeness is to compute all possible narrowing steps of a term, but in most cases this would be quite inefficient since many of these steps would be unnecessary.

Similar to 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 of interest 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. Figure 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* [20] implicitly define a corresponding function over algebraic data types such as those of

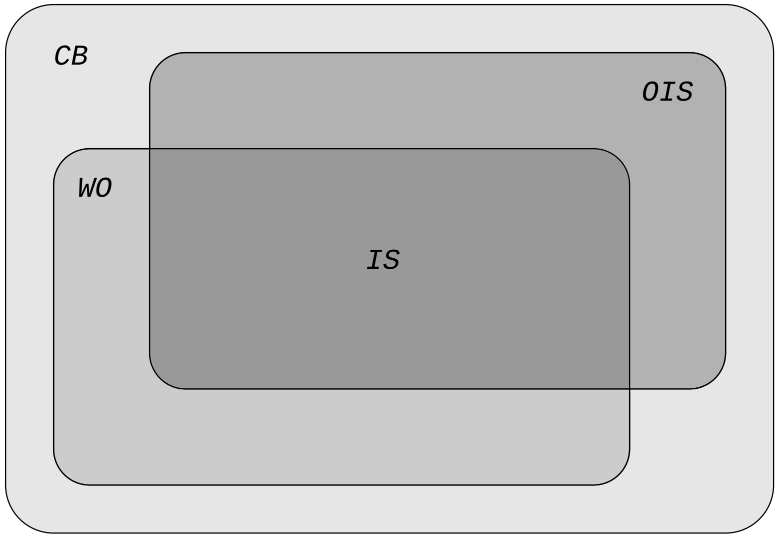


Fig. 1. Containment diagram of rewrite systems modeling functional logic programs. The outer area, labeled *CB*, represents the *constructor-based* rewrite systems. The smallest darkest area, labeled *IS*, represents the *inductively-sequential* rewrite systems. These are the intersection of the *weakly-orthogonal*,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

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 function logic, programming. The following section will relax this limitation. The discussion of this section is also limited to unconditional TRSs. Constructor-based TRSs can be transformed into unconditional TRSs by a transformation that preserves both values and computations without loss of either efficiency or generality. This also will be addressed in the next section.

3.1 Inductively Sequential TRSs

The smallest class in the diagram of Figure 1 is the *inductively sequential* TRSs. These are the strongly sequential component of the constructor-based TRSs [14]. Optimal rewrite derivations for the strongly sequential TRSs are executed by the well-know *call-by-need* strategy [16]. Optimality, in this class, is the property that every step of a call-by-need derivation is *needed* in the sense that the value computed by the derivation, if it exists, cannot be reached unless the step is executed. *Needed narrowing* [6] is a conservative extensions of this strategy, i.e., rewrite derivations executed by needed narrowing are call-by-need derivations. 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 pair-wise disjoint [6, Def. 15]. 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.

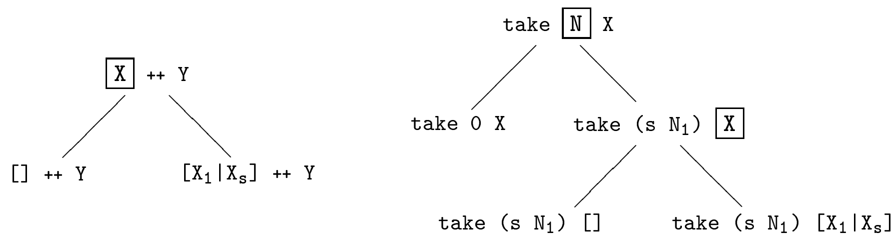
The inductively sequential TRSs were initially characterized through the concept of definitional tree [2]. Since definitional trees are frequently used to define and implement narrowing strategies for several subclasses of the constructor-based TRSs, I recall this concept. A *definitional tree* of an 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 same parent differ from each other only at the position, referred to as *inductive*, of a variable of their parent.

Example 5. Consider an operation, `take`, that returns a prefix of a list. For the purpose of this discussion, the natural numbers are represented in Peano notation.

```
data nat = 0 | s nat
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.



A TRS is inductively sequential iff all its operations have a definitional tree. Needed narrowing is defined through definitional trees that are used as finite state automata to compute narrowing steps. I give an informal account of this computation in an example. The formal definition is in [6, Def. 13].

Example 6. Needed narrowing computes a step of a term t rooted by `take`, i.e., $t = \texttt{take}\ n\ x$, as follows. Let π be an element of the definitional tree of `take` that 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 all this more concrete, suppose that $t = \texttt{take N [1]++[2]}$, where `N` is a free variable. Term t unifies with both `take 0 X`, which is a leaf, and `take (s N`$_1$`) X`, which is a branch. Therefore, needed narrowing computes the two steps shown below. Each steps is shown with its substitution.

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

Observe that the substitution of the second step is not most general. This characteristic of needed narrowing is a major departure from previously proposed strategies. Unless `N` is instantiated to (`s N`$_1$), the step could turn out to be useless, e.g., when followed by a step instantiating `N` to `0`.

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). Rules's left-hand 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, although this class admits sequential normalizing rewrite strategies, as well. Optimal rewrite derivations for the weakly orthogonal constructor-based TRSs are executed by repeatedly contracting all the redexes of a *necessary set* [22]. This notion generalizes that of needex redex, which is undefined in this class. Optimality, in this class, is the property that the value computed by a derivation, if it exists, cannot be reached unless one redex in a necessary set is contracted. In general, no efficient procedure is known to determine this redex.

Example 7. An emblematic non-inductively sequential operation in this class is the *parallel-or* defined by the rules:

`or true _ -> true`	R_6
`or _ true -> true`	R_7
`or false false -> false`	R_8

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

Weakly needed narrowing [5] is a conservative extension of the strategy defined in [22]. This strategy is formulated by means of definitional trees as well.

The rewrite rules defining an operation in a weakly orthogonal TRSs can be partitioned into inductively sequential subsets, i.e., subsets for which there exists a definitional tree. For the rules of Example 7, one such partition is $\{R_6, R_8\} \uplus \{R_7\}$. Then, a necessary set of redexes is obtained by computing a needed redex for each element of a partition. This rewrite strategy, formalized in [2], is equivalent to [22]. Its extension to narrowing is straightforward, but the properties of a necessary set of narrexes differ from those of a necessary set of redexes. The narrowing step computed by an element of the partition of the rewrite rules may have a substitution incompatible with that of another step and/or the position of a step may not be disjoint from that another step. Neither condition may occur for rewrite steps.

Several related narrowing strategies dealing with these conditions are discussed in [5], but none claims the strong optimality results of [22]. However, all these strategies are optimal for rewrite derivations, since they compute the same steps as [22].

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. Rewrite rules in overlapping inductively sequential TRSs can overlap, but only if their left-hand sides are equal modulo a renaming of variables. By contrast to the weakly orthogonal TRSs, no restriction is placed on the right-hand sides of overlapping rewrite rules. Computations in this class are sometimes referred to as non-deterministic.

Example 8. 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 ++ "|" ++ regexp X
```

The definition of operation `regexp` closely resembles the formal definition of *regular expression*. Non-deterministic operations contribute to the expressive power of a language. For example, to recognize whether a string, say s, denotes a well-formed regular expression over the alphabet of digits it simply suffices to evaluate (`regex 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.

The evaluation strategy for overlapping inductively sequential TRSs is *INS* [3]. This strategy has been formulated for narrowing computations since its inception, i.e., it does not originate from an earlier rewrite strategy. In this class, every term that can be narrowed to a value has a needed narrex. Since there may exist several rewrite rules with the same left-hand side, a narrex may have several replacements. Optimality, in this class, is the property that every step of an *INS* derivation is *needed* in the sense that the value computed by the derivation, if it exists, cannot be reached unless the narrex is contracted. However, not every replacement of a needed narrex is needed, hence *INS* makes needed steps modulo non-deterministic choices. In general, no efficient procedure is known to determine which choices of replacements are needed.

Non-deterministic operations require re-thinking some semantic aspects of both evaluation and strategies. For example, the meaning of the "`=`" operation is generalized to *joinability*, i.e., $t = u$ means that t and u have a common value — one out of possibly many. Another relevant issue is the step of a derivation in which a *value* is eventually bound to a variable. This is a subtle point, since the value bound to the variable needs not be computed at that step. Two practical examples clarify the issue.

Operation `queens`, defined in the introduction, has a rule with three occurrences of variable `Y`. Variable `Y` is initially bound to `permute X`, which may eventually be reduced to one of many values. Replacing each occurrence of `Y` with `permute X` and evaluating each occurrence independently would be clearly incorrect. The *value* of the occurrence returned by operation `queens` could differ from that tested for safety using operation `capture`. In this case, the intended behavior, called *call-time* choice semantics, is to bind the same value to all the occurrences of `Y`.

Operation `regexp`, defined in this section, has rules with two occurrences of variable `X`. Variable `X` is initially bound 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. Unless 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, called *need-time* choice semantics, is not to bind the same value to all the occurrences of `X`.

In each case, the intended behavior depends on the program. A functional logic language should allow the programmer to encode in a program the appropriate semantics. A strategy for non-deterministic computations should have useful properties, e.g., soundness and completeness, for both semantics.

3.4 Constructor-based TRSs

The constructor-based TRSs are the largest class that has been proposed for modeling functional logic programs. They are a proper superclass of all the other classes discussed previously. Overlapping of rules's left-hand sides is unrestricted, though in constructor-based TRSs it may occur only at the root. No specific restrictions are imposed on the right-hand sides of overlapping rules.

Example 9. The following definition of operation `permute` is an alternative to that proposed in the *N-queens* program. 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]
```

An early narrowing strategy for this class is presented in [18]. That strategy is a generalization to narrowing of a rewrite strategy proposed in [1]. The completeness of both these strategies is unknown.

A potential difficulty of a class as large as the constructor-based TRSs is that outermost rewrite strategies are not normalizing [4], 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 proving their properties, e.g, completeness and optimality. For example, consider the evaluation of t = `insert u v`. One cannot tell whether position 2 of t is needed. In fact, one must evaluate subterm v to apply one rewrite rule of `insert`, but not apply the other rewrite rule. Both [1] and [18] are *demand-driven*, rather than *needed*, strategies 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. 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 can also be quite inefficient when the application of a rule to a term and/or the evaluation of a subterm for the application of a rule are unnecessary.

Very recently, a transformational approach has been proposed [4] for functional logic computations in the constructor-based TRSs. This approach transforms a constructor-based TRS, $\mathcal{R}$, into an overlapping inductively sequential TRS, $\mathcal{R}'$. Computations in $\mathcal{R}'$ are executed by *INS* which sound, complete and efficient. The transformation itself is sound and complete in the following sense. The transformation adds new operation symbols, but no new constructors, to the signature of $\mathcal{R}'$. The change in signature generally creates new steps and new normal forms. However, 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 $\mathcal{R}'$. 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 $\mathcal{R}$.

4 Related issues

The previous sections have almost entirely neglected some important issues related to functional logic computations. I briefly address these issues in this section. The focus, as in the rest of this paper, is on strategies.

The classes of TRSs discussed earlier are all unconditional. The well-known outermost-fair rewrite strategy, which is normalizing for almost orthogonal TRSs [20], is also normalizing for conditional almost orthogonal TRSs [9]. For the constructor-based TRSs, the results presented earlier about evaluation strategies are extended to the conditional case with little effort. The strategies discussed in Section 3 are based, either directly or indirectly, on definitional trees. Definitional trees are concerned with 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 transforms an *original* conditional TRS into a *target* unconditional one without altering the left-hand sides of rewrite rules. In this way, results proved for the target TRS are transferred to the original TRS. This transformational approach is formalized in [4]. In short, the condition of a conditional rewrite rule is moved into the right-hand side by introducing a conditional operation. More precisely, a conditional rewrite rule of the form:

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

is transformed into:

$$l \to \textbf{if}\; t_1 = u_1, \; \ldots, \; t_n = u_n \;\textbf{then}\; r$$

where, as expected, the **if** $\cdot$ **then** $\cdot$ binary operation returns its second argument when its first argument succeeds. The introduction of this new operation generally creates new steps and new normal forms, but not new values. The computations to a value with the transformed rewrite rules remain essentially the same.

A second relevant issue about functional logic programming 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 [20], is normalizing also for weakly-orthogonal higher-order TRSs if an additional condition, full extension, is imposed on higher-order rewrite rules [23]. 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 [12], *applicative* TRSs [19], and *higher-order* inductively sequential TRSs [15]. 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-know transformation for higher-order logic computations [24]. 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 a 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 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 `s` is the constructor defined in Example 5. Most current 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, 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 [24] is formulated for functional evaluation in logic programming, [12] generalizes it to narrowing, and [7] 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 contains an overview of evaluation strategies for functional logic programs. A program is (seen as) a constructor-based TRSs and an evaluation or computation is a rewriting or 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. When presented, the rewrite strategy is normalizing, i.e., it computes the value, if it exists, of a term. In addition, the narrowing strategy is sound and complete, i.e., when used to solve an equation it computes only and all the equation's solutions. All these strategies are also, to varying degrees, theoretically efficient. Not surprisingly, as classes get bigger the claims about the efficiency of strategies used 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. Evaluation strategies for these extensions are not as well developed as for the ordinary case. Transformations from extended TRSs to ordinary TRSs make it possible to use the strategies presented earlier while preserving many of their most desirable properties.

References

1. S. Antoy. Non-determinism and lazy evaluation in logic programming. In T. P. Clement and K.-K. Lau, editors, *Logic Programming Synthesis and Transformation (LOPSTR'91)*, pages 318–331, Manchester, UK, July 1991. Springer-Verlag.
2. S. Antoy. Definitional trees. In *Proc. of the 4th Intl. Conf. on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.
3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of the 6th International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
4. S. Antoy. Constructor-based conditional narrowing, 2001. Submitted for publication. Available at URL http://www.cs.pdx.edu/~antoy/.
5. S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. of the 14th International Conference on Logic Programming (ICLP'97)*, pages 138–152. MIT Press, 1997.
6. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, July 2000.
7. S. Antoy and A. Tolmach. Typed higher-order narrowing without higher-order strategies. In *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722, pages 335–350, Tsukuba, Japan, 11 1999. Springer LNCS.
8. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
9. J. A. Bergstra and J. W. Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986.
10. R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, New York, NY, 1988.
11. N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North Holland, Amsterdam, 1990.
12. J. C. González-Moreno. A correctness proof for Warren's HO into FO translation. In *Proc. GULP' 93*, pages 569–585, Gizzeria Lido, IT, Oct. 1993.
13. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *The Journal of Logic Programming*, 19&20:583–628, 1994.
14. M. Hanus, S. Lucas, and A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8, 1998.
15. M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. In *Proc. 7th International Conference on Rewriting Techniques and Applications (RTA'96)*, pages 138–152. Springer LNCS 1103, 1996.

16. 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*, pages 395–443. MIT Press, Cambridge, MA, 1991.
17. 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.
18. R. Loogen, F. López Fraguas, and M. Rodríguez Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. 5th International Symposium on Programming Language Implementation and Logic Programming (PLILP'93)*, pages 184–200. Springer LNCS 714, 1993.
19. K. Nakahara, A. Middeldorp, and T. Ida. A complete narrowing calculus for higher-order functional logic programming. In *Proc. 7th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'95)*, pages 97–114. Springer LNCS 982, 1995.
20. M. J. O'Donnell. *Computing in Systems Described by Equations*. Springer LNCS 58, 1977.
21. R. A. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
22. R. C. Sekar and I. V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. *Information and Computation*, 104(1):78–109, 1993.
23. F. van Raamsdonk. Higher-order rewriting. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA '99)*, pages 220–239. Springer LNCS 1631, 99.
24. D. H. D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.