Constructor-based Conditional Narrowing*

Sergio Antoy
Dept. of Computer Science
Portland State University

antoy@cs.pdx.edu

ABSTRACT

We define a transformation from a left-linear constructor-based conditional rewrite system into an overlapping inductively sequential rewrite system. This transformation is sound and complete for the computations in the source system. Since there exists a sound and complete narrowing strategy for the target system, the combination of these results offers the first procedure for provably sound and complete narrowing computations for the whole class of the left-linear constructor-based conditional rewrite systems. We address the differences between demand driven and lazy strategies and between narrowing strategies and narrowing calculi. In this context, we analyze the efficiency and practicality of using our transformation for the implementation of functional logic programming languages. The results of this paper complement, extend, and occasionally rectify, previously published results in this area.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.1.6 [**Programming Techniques**]: Logic Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures*; D.3.4 [**Programming Languages**]: Processors—*Optimization*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems; I.1.1 [**Algebraic Manipulation**]: Expressions and Their Representation—*Simplification of expressions*; I.2.2 [**Automatic Programming**]: Program transformation

General Terms

Algorithms, Languages, Performance, Theory

Keywords

Functional Logic Programming Languages, Rewrite Systems, Narrowing Strategies, Call-By-Need

*This work has been supported in part by the National Science Foundation grant INT-9981317.

Third International Conference on Principles and Practice of Declarative Programming (PPDP'01), Firenze, Italy, Sept. 2001, pages 199–206.

1. INTRODUCTION

Functional logic programs are modeled, in large part, by rewrite systems (TRSs). A key decision in the design of functional logic languages is the class of TRSs chosen to model the programs. In principle, power and generality are very desirable. In practice, extreme power or the greatest generality are not necessarily a good thing. To some extent, the situation that we are discussing parallels the use of the *goto* statement in imperative languages. While the *goto* statement offers the greatest control over the flow of execution, its use has been considered "harmful." In fact, a recently designed, very successful imperative language has completely eliminated it in favor of a more disciplined approach.

In the case of functional logic programming, the problem of a large class of TRSs is the difficulty of both discovering a sound and complete narrowing strategy and implementing it efficiently. We limit the discussion to first-order computations and the focus to strategies. Figure 1 summarizes the state of the art in this field. All the TRSs in the figure are *constructor-based* (follow the constructor discipline [21]) and *left-linear*. For the time being, we ignore whether they are *conditional* [8]. We will see later that this is not a substantial characterization for our discussion.

The inductively sequential TRSs are the first-order component of functional languages, such as *ML* and *Haskell*, and the constructor-based restriction of the strongly sequential TRSs [14, 15]. A sound and complete strategy for this class is *needed narrowing* [5]. The needed narrowing strategy extends to narrowing the well-known *call-by-need* optimal rewrite strategy [15]. The extension, which is conceptually surprisingly simple, preserves the optimality of [15] for rewriting and extends it to the independence of computed answers [5].

The weakly orthogonal TRSs include operations that express *or parallelism*. In this class, computations yield deterministic results, but have no simple sequential strategy. A sound and complete strategy for this class is *parallel narrowing* [4]. Parallel narrowing extends to narrowing an optimal rewrite strategy [22]. The extension to narrowing, which is unexpectedly complicated, preserves the optimality of [22] for rewriting, but does not fully extend it to narrowing.

The overlapping inductively sequential TRSs are interesting in functional logic programming because they express non-deterministic computations, i.e., a same term can have distinct normal forms. By contrast to the more traditional use of predicates for this purpose, non-deterministic operations can be functionally composed and computations, including those containing non-deterministic operations, can be lazily executed. A sound and complete strategy for this class is *INS* [3]. *INS* does not originate from a previous rewrite strategy. *INS* conserves the simplicity of needed narrowing and extends some of its optimality results modulo non-deterministic

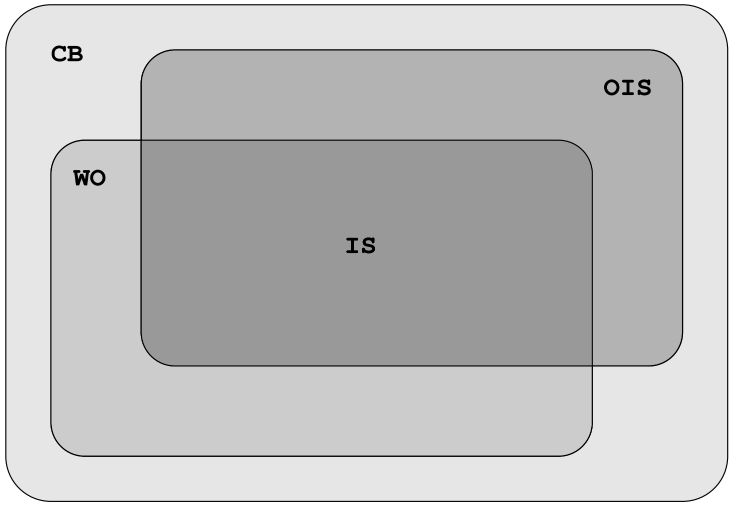

Figure 1: Containment diagram of rewrite systems modeling functional logic programs. The outer area, labeled *CB*, represents the *constructor-based* rewrite systems. The smallest darkest area, labeled *IS*, represents the *inductively-sequential* rewrite systems. These are the intersection (rules differing for a renaming of variables are considered equal) of the *weakly-orthogonal*, labeled *WO*, and the *overlapping inductively-sequential* rewrite systems, labeled *OIS*.

choices.

The situation becomes cloudier for the whole class of the constructor-based TRSs. A strategy for this class is proposed in [1] for non-deterministic functional computations in a logic programming language. A generalization of [1] to narrowing computations and conditional rewrite rules is proposed in [17], but the presentation is limited to confluent TRSs and thus it excludes non-determinism. Both strategies are based on structurally equivalent definitional trees, thus we believe that they compute the same steps. Little is known about the properties of these strategies. Probably, both are sound and complete for the whole class of the constructor-based TRSs. These strategies are *demand driven*, rather than *lazy*. A *calculus*, rather than a *strategy*, is also known [11] for the constructor-based TRSs. This calculus is sound and complete for *the solution of admissible goals*.

This short summary shows that an important result is still missing, namely, a provably sound and complete narrowing strategy for the constructor-based TRSs. The missing result is the contribution of this paper.

In Section 2 we briefly recall background information and foundations for understanding our results. In Section 3 we discuss in some details the motivation of our work. In Section 4 we define a transformation of TRSs. We transform a constructor-based conditional TRS, $\mathcal{R}$, into an overlapping inductively sequential TRS, $\mathcal{R}'$. Our transformation is sound and complete for the computations of $\mathcal{R}$. Thus, using *INS* in $\mathcal{R}'$ we obtain a sound and complete narrowing procedure for $\mathcal{R}$. Section 5 contains our conclusions. The Appendix sketches the proof of a non-trivial result claimed in Theorem 1.

2. FOUNDATIONS

2.1 Background

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, but for the sake of simplicity we restrict our presentation to the unsorted case. Signature Σ is partitioned into a set $\mathcal{C}$ of *constructor* symbols and a set $\mathcal{D}$ of *defined operations* or functions. The set of *terms* is constructed over Σ and a countably infinite set $\mathcal{X}$ of *variables*. The set of *values* is constructed over $\mathcal{C}$ and $\mathcal{X}$. We will see shortly that every value is a normal form, but some normal forms may not be values. A *pattern* is a term $f(t_1, \ldots, t_n)$ in which $f \in \mathcal{D}$ 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. As usual, we require that any variable in r must occur in l, i.e., $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. However, this limitation which we will revisit later is unnecessarily restrictive for narrowing computations. An unconditional TRS, $\mathcal{R}$, defines a rewrite relation $\to_{\mathcal{R}}$ on terms as follows: $s \to_{\mathcal{R}} t$ if there exists a rewrite rule $l \to r$ in R, a substitution σ and a context C such that $s = C[l\sigma]$ and $t = C[r\sigma]$.

A *conditional* rewrite rule is a triple $l \to r \Leftarrow c$, where l and r are defined as in the unconditional case and c is a *sequence of conditions*. Various options have been considered for the form of the conditions [8]. Since we are interested in narrowing computations in possibly non-terminating constructor-based TRSs, it is appropriate to consider conditions consisting of sequences of *elementary equational constraints*, i.e., pairs $u \approx v$, where the symbol "$\approx$" is an ordinary (infix, overloaded) operation, called *strict equality*, defined for every constant constructor c and constructor d of arity $n > 0$ by the rules:

$$\begin{array}{l} c \approx c \to \texttt{success} \\ d(u_1, ..., u_n) \approx d(v_1, ..., v_n) \to u_1 \approx v_1 \;\texttt{\&}\; \ldots \;\texttt{\&}\; u_n \approx v_n \end{array} \tag{1}$$

where success is a new constructor symbol and & is a new infix operation defined by the rule:

$$\texttt{success \& X} \to \texttt{X} \tag{2}$$

We call a *TRS with equality* any TRS containing the strict equality equations, (1), and their associated symbols and equation, (2). It is easy to verify that an elementary equational constraint $t \approx u$ evaluates to success if and only if t and u evaluate to a same *value*.

In the terminology of [8], the TRSs that we consider are similar to type I, the difference being that in a type I TRS the sides of a condition must evaluate to a common reduct, whereas we additionally require that the common reduct is a value. With a trivial syntactic change, we can also regard our TRSs as type III_n by interpreting $u \approx v$ as $u \approx v \xrightarrow{*} \texttt{success}$.

The definition of the rewrite relation for conditional TRSs is fairly more complicated than for unconditional TRSs. We refer to [8] for a classic approach. We say that s *narrows* to t with substitution σ, denoted $s \rightsquigarrow_\sigma t$, if σ is an idempotent constructor substitution such that $s\sigma \to t$. It is usually required that σ is a most general unifier, but we waive this restriction because narrowing with most general unifiers is suboptimal [5] (for a direct example see [3, Ex. 12]). With this definition, narrowing is trivially complete, but less operational. Computing narrowing steps is the task of a *strategy*, the subject of the next section. A *computation* or *evaluation* of a term s is a narrowing derivation $s \overset{*}{\rightsquigarrow}_\sigma t$, where t is a value. Substitution σ is called a *computed answer* and t is called a value or a *computed expression* of s.

2.2 Strategies and Calculi

A *narrowing strategy* is a partial mapping from terms to narrowing steps. A key attribute of a strategy is "*laziness*". This word has been used with different meanings. *Lazy* functional languages compute with infinite data structures because they execute only steps that are *needed* — which means that if they are avoided, computations may no longer be complete. The formalization of "need" is subtle and depends on the class of TRSs operated upon by the strategy. E.g., this notion for the inductively sequential TRSs is stronger than for larger classes. In functional logic programming, "*laziness*" has been used (incorrectly in our opinion) in a weaker sense that does not preserve the meaning of the word in functional programming. Sometimes, "*lazy*" has been used as a synonym of "demand driven", a concept weaker than "needed", which will be explained shortly. In the rest of this paper, we use this word in the stricter sense.

For both rewriting and narrowing, the various definitions of *need* are based on whole computations, e.g., see [22, Def. 6] and [5, Def. 10]. The rewrite rules obviously must play a role in determining whether a step is needed, but are not directly referenced by the definition of needed step. By contrast, a *demand driven* [10, p. 179] strategy determines whether to execute a step from the rewrite rules, often from the rules' left-hand sides only, e.g., see [1, 17]. In this case, properties of the entire computation play little or no role in the definition of the strategy. Thus, a step executed by a demand driven strategy is needed for the application of a rule to a term, but the application of the rule may not be needed for computing the value of the term. A contrived instance of this situation is presented in Example 3. A less contrived instance is in [3, Ex. 12].

Narrowing *calculi*, e.g., LCN [20] for confluent TRSs, OINC [16] for orthogonal TRSs and goals whose right-hand side is a ground normal form, CLNC [11] for left-linear constructor-based TRSs, have been investigated as alternatives to narrowing *strategies*. A common reason advocated to study a calculus rather than a strategy is that "narrowing is a complicated operation" [19]. 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., $\mathrm{LCN_d}$ [19] for left-linear confluent constructor based TRSs with strict equality, and S-OINC [16].

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 we would guess that, in general, strategies can be implemented more easily and efficiently than calculi because strategies more directly relate to the terms that are the object of a computation.

2.3 Semantic options

Two semantic options have been considered for non-deterministic computations. One, referred to as *call-time choice*, fixes the values of terms at the time of a "call". The other, referred to as *run-time choice*, is the usual rewrite semantics. Perhaps, a better name would be *need-time choice*, since both choices are executed at run-time.

The two semantics differ only when a non right-linear rewrite rule creates several needed descendants of a same term with distinct reducts. The call-time choice reduces all the descendants to a same reduct, whereas the need-time choice may reduce distinct descendants to distinct reducts. Both semantics are easy to implement. The call-time choice is obtained by sharing the representation of the descendants of term. The need-time choice, being ordinary rewriting, does not require any device.

The call-time choice has been advocated as the "most natural way to combine non-deterministic choices with parameter passing" [11]. Indeed, in many cases, this is the only appropriate semantics. However, we think that a functional logic language should not impose either choice, since there exist situations where the call-time choice is totally inappropriate. An example of this situation is presented in the next section. Therefore, syntactic constructs in a functional logic programming language should allow the programmer to choose either semantics depending on a program or selected portions of a program.

2.4 INS

The last crucial component needed to understand our work is the *Inductively Sequential Narrowing Strategy*, or *INS*. This strategy is defined for the class of the *overlapping inductively sequential* TRSs [3]. Each operation in this class has left-hand sides that can be organized in a definitional tree, but several distinct right-hand sides are allowed for a same left-hand side. Operations defined by rewrite rules of this kind support an expressive and natural formulation of problems into programs.

Example 1. A program to generate (non-empty) regular expressions over a given alphabet is the direct encoding of its specification[1]. Using the notation of [3]:

```
regexp X → X
         | "(" ++ regexp X ++ ")"
         | regexp X ++ regexp X
         | regexp X ++ "*"
         | regexp X ++ "|" ++ regexp X
```

where `++` denotes concatenation. The following expression:

`regexp ("0" | "1")` $\approx t$

evaluates to `success` if and only if t is a well-formed regular expression over the alphabet $\{\texttt{0}, \texttt{1}\}$, and thus it acts as a (rather rudimentary) parser.

The strategy for computations in this class of TRSs is *INS* [3], a conservative extension of *needed narrowing* [5]. Similar to needed

[1]The grammar underlying this specification is ambiguous and ignores the usual operator precedence. These aspects are irrelevant to our discussion.

narrowing, *INS* is sound, complete and efficiently implemented. It preserves some of needed narrowing's properties, such as a slightly weaker notion of needed redex, but it does not preserve the optimality of the number of steps or the disjointness of computed answers. These desirable properties are lost due to the *don't know* non-deterministic choices inherent to a problem, hence the loss appears to be unavoidable. One could argue that *INS* performs the best possible job under its intended conditions of employment.

3. MOTIVATION

Narrowing in left-linear constructor-based conditional TRSs is discussed in [11]. We find the treatment in Section 8 of [11], which deals with the "practicability of the [CRWL] approach," unsatisfactory. There, the authors acknowledge a "big gap" between their presentation of a lazy narrowing calculus, CLNC, and the implementation of a programming language. Supposedly, this gap is to be filled by a "suitable narrowing strategy." More specifically, the authors of [11] "strongly conjecture" the existence of "a theoretical result that would guarantee soundness and completeness of a demand driven strategy w.r.t. CRWL semantics" and "taking the previous conjecture for granted, ... that any implementation of the demand driven strategy (as presented in Ref. [17]) can be safely used for the execution of CRWL-programs."

Unfortunately, the presentation in [17] does not prove that the demand driven strategy is sound, complete and/or efficient. We do believe that this strategy is sound and complete. A proof of these properties should be eased by this paper. We have reservations on the efficiency of this strategy. More generally, we doubt the existence of sound and complete narrowing strategies for the constructor-based TRSs which can be implemented with an efficiency competitive w.r.t. lazy strategies. The lack of fundamental results about the demand driven strategy presented in [17] is compounded by some errors or imprecisions occurring in [11].

It is claimed [11, p. 82] that "CLNC-derivations proceed by outermost narrowing." We refute this claim since it is inconsistent with the completeness of CLNC [11, Th. 7.2]. Outermost strategies are incomplete for constructor-based TRSs. The operations occurring in the following counterexample are all defined in [11, pp. 53-54].

Example 2. Consider all the outermost derivations of:

$$\texttt{sorted(permute([1,2]))} \qquad (t_0)$$

The only available step of t_0 yields:

$$\texttt{sorted(insert(1,permute([2]))} \qquad (t_1)$$

There exists a unique outermost step of t_1 which yields:

$$\texttt{sorted([1|permute([2])])} \qquad (t_2)$$

No matter what the remaining steps of a derivation of t_2 are, term `false` can no longer be reached from t_2. However, it is easy to verify that `false` can be reached from t_0.

We also dispute the use of the adjective "lazy" in CLNC.

Example 3. Consider the following left-linear constructor-based TRS:

$$\begin{array}{l}\texttt{f(c)} \to \texttt{c}\\ \texttt{f(X)} \to \texttt{c}\\ \texttt{g} \to \ldots\end{array}$$

and the term $t = \texttt{f(g)}$. Should subterm `g` of t be evaluated? Needed strategies, such as those discussed in [2], don't, but demand driven strategies, such as those discussed in [1, 17], and CLNC do.

In more general situations the discussion becomes even more complicated. We are not aware of any meaningful notion of laziness for computations in the constructor-based TRSs. We believe that the laziness of a strategy, or a calculus, for this class of TRSs is only wishful thinking.

While both of the above imprecisions are easy to fix—one can simply drop the claims of outermostness and laziness—they are symptomatic of a bigger problem. Calculi are not always well-suited for an implementation, and the details of narrowing, in particular strategies, are complicated enough to deserve a careful investigation. This is the thrust of this paper.

4. TRANSFORMATIONS

In this section we present a transformation of TRSs and show how it solves the problem at hand. The source TRS, $\mathcal{R}$, is left-linear, constructor-based and conditional. The target TRS, $\mathcal{R}'$, is overlapping inductively sequential. We show that any *useful* computation in $\mathcal{R}$ has a corresponding computation in $\mathcal{R}'$. A sound, complete and lazy strategy, *INS* [3], is available for $\mathcal{R}'$. Therefore, we provide all the components necessary to fill the "big gap" mentioned in [11, Sect. 8].

4.1 Linearization

We consider exclusively *left-linear* TRSs. The reader might wonder whether this is a sensible or altogether necessary limitation. Below, using an example, we show how we could transform a non left-linear TRSs into left-linear one with *similar* meaning.

Example 4. Consider the following (non left-linear) rewrite rule:

$$\texttt{f(X,X)} \to t$$

where `f` is an operation and t is some term. According to this rule, the term $\texttt{f}(u\texttt{,}v)$ should be contracted when u and v are "equal". There are two potential difficulties with this statement: the precise meaning of "equal" and the degree to which u and v should be evaluated to determine whether they are equal.

The rule offers no hints on how much to evaluate each argument of `f`. If at all, it seems to suggest not to evaluate them. Furthermore, the ordinary rewrite semantics would suggest to use unification for testing the equality of the arguments. In functional logic programming with non-strict semantics, the validity of an equation is defined as strict equality on terms. This is the preferred criterion of equality for lazy computations in non-terminating TRSs. Therefore, a term such as $\texttt{f}(u\texttt{,}u)$ should not be contracted unless u evaluates to a *value* even if the term matches the rule's left-hand side.

Both the problems discussed above are eliminated by the following rule:

$$\texttt{f(X,Y)} \to t \Leftarrow \texttt{X} \approx \texttt{Y}$$

According to this rule, the term $\texttt{f}(u\texttt{,}v)$ is contracted when u and v are "strictly equal," i.e., they evaluate to a same value. Furthermore, the rules of $\approx$, together with a sensible strategy, control the evaluation of the arguments of `f` to determine whether they are equal.

Thus, we could transform non left-linear TRSs into left-linear ones as sketched by the above example since the left-linearized rewrite rules capture the exact semantics that we desire. The potential problem would be that the transformation would lose the ordinary semantics of non left-linear rules. Therefore, it seems more appropriate to exclude non left-linear TRSs from our treatment because their semantics is inappropriate for functional logic programming. It should be obvious from this discussion that imposing left-linearity does not entail any loss of computing power or program expressiveness.

4.2 Deconditionalization

Our first task is to get rid of the conditions. Unconditional TRSs are somewhat simpler than conditional ones. This has lead to various proposals of transformations of conditional TRSs into unconditional ones, e.g., [8, 12, 18]. The interest in conditional TRSs is that some syntactic restrictions, e.g., type III_n [8], of the conditions of the rewrite rules ensure confluence. The TRSs we consider, and in particular both the left-linear constructor-based TRSs and the overlapping inductively sequential TRSs, are not confluent. Thus, we have much more leeway in eliminating the conditions since we need to preserve computations, but not confluence.

Definition 1. [Deconditionalization] If $\mathcal{R} = \langle \Sigma, R \rangle$ is a left-linear constructor-based conditional TRS with equality, we define a new TRS, $\mathcal{R}' = \langle \Sigma', R' \rangle$, called the *deconditionalization* of $\mathcal{R}$. We introduce a new overloaded binary operation denoted by `if`, i.e., $\Sigma' = \Sigma \cup \{\mathtt{if}\}$. The set of rules R' is defined by:

1. R' contains every unconditional rule of R
2. R' contains the rule
   $\mathtt{if(success,X)} \to \mathtt{X}$
3. If $l \to r \Leftarrow c$ is a conditional rule of R, R' contains the rule
   $l \to \mathtt{if}(c,r)$
4. R' contains no other rule

The following statements relate terms, values, rewrite relation, and computations between a TRS and its deconditionalization.

THEOREM 1. *Let $\mathcal{R}$ be a left-linear constructor-based TRS with equality and $\mathcal{R}'$ its deconditionalization.*

1. *$\mathcal{R}'$ is a left-linear constructor-based unconditional TRS with equality*
2. *The set of terms of $\mathcal{R}$ is contained in the set of term of $\mathcal{R}'$*
3. *The set of values of $\mathcal{R}$ is the same as the set of values of $\mathcal{R}'$*
4. *If t and v are respectively a term and a value of $\mathcal{R}$ (and by the previous points, of $\mathcal{R}'$), then $t \xrightarrow{*}_{\mathcal{R}} v$ if and only if $t \xrightarrow{*}_{\mathcal{R}'} v$*

The most interesting consequence of the above statements is that although both the set of terms and the rewrite relation in $\mathcal{R}$ may differ from those in $\mathcal{R}'$, the rewrite computations of $\mathcal{R}$ can be simulated by those of $\mathcal{R}'$ because we care only about derivations that terminate in a value. The same holds true for narrowing computations since the narrowing relation is sound and complete w.r.t. the rewrite relation.

Theorem 1 allows us to restrict our attention to unconditional TRSs only. From a theoretical standpoint, there is no gain in computational power or expressiveness by considering conditional rewrite rules. From a practical standpoint, the syntax of a programming language may preserve conditional rules if programmers find them familiar or expressive enough.

4.3 Sequentialization

We now define the transformation from a constructor-based (unconditional) TRS into an overlapping inductively sequential TRS. We observe that the constructor discipline "localizes" to single operations many fundamental properties of TRSs. For example, a TRS $\mathcal{R}$ is overlapping or weakly orthogonal or inductively sequential if and only if the set of rewrite rules defining each operation of $\mathcal{R}$ is respectively overlapping or weakly orthogonal or inductively sequential. This property allows us to define our transformation for a single operation rather than for an entire TRS.

Definition 2. [f-sequentialization] Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a left-linear constructor-based TRS and f a defined operation of $\mathcal{R}$. We define a new TRS, $\mathcal{R}' = \langle \Sigma', R' \rangle$, called the *$f$-sequentialization* of $\mathcal{R}$. If f is overlapping inductively sequential, then $\mathcal{R}' = \mathcal{R}$. Otherwise, let R_f be the set of the rewrite rules defining f in $\mathcal{R}$ and let $R_1 \uplus R_2 \uplus \cdots \uplus R_n$ be a partition of R_f such that for all $i = 1, \ldots, n$, R_i is an overlapping inductively sequential set of rewrite rules. Let R'_i be the set of rewrite rules obtained from R_i in which the occurrence of f in the left-hand side is replaced by a new symbol denoted by f_i. Let ρ_f denote the following rewrite rule

$$\begin{array}{rl} f(X_1, \ldots, X_k) \to & f_1(X_1, \ldots, X_k) \\ | & \cdots \\ | & f_n(X_1, \ldots, X_k) \end{array}$$

Let $R'_f = \{\rho_f\} \cup R'_1 \cup \cdots \cup R'_n$. TRS $\mathcal{R}'$ is defined by $R' = (R - R_f) \cup R'_f$ and $\Sigma' = \Sigma \cup \{f_1, \ldots, f_n\}$.

Example 5. The following TRS defines operation `insert` also found in [11]:

```
insert(X,Ys) → [X|Ys]
insert(X,[Y|Ys]) → [Y|insert(X,Ys)]
```

Operation `insert` is not overlapping inductively sequential. The `insert`-sequentialization of the above rules is:

```
insert(X,Ys) → insert1(X,Ys) | insert2(X,Ys)
insert1(X,Ys) → [X|Ys]
insert2(X,[Y|Ys]) → [Y|insert(X,Ys)]
```

Observe that every operation in the `insert`-sequentialized TRS is overlapping inductively sequential (it is defined by a single rewrite rule).

The following statements, analogous to those of Theorem 1, relate terms, values, rewrite relation, and computations between a TRS and one of its sequentializations.

THEOREM 2. *Let $\mathcal{R}$ be a left-linear constructor-based TRS, f a defined operation of $\mathcal{R}$ and $\mathcal{R}'$ the f-sequentialization of $\mathcal{R}$.*

1. *$\mathcal{R}'$ is a left-linear constructor-based TRS*
2. *The set of terms of $\mathcal{R}$ is contained in the set of term of $\mathcal{R}'$*
3. *The set of values of $\mathcal{R}$ is the same as the set of values of $\mathcal{R}'$*
4. *If t and v are respectively a term and a value of $\mathcal{R}$ (and by the previous points, of $\mathcal{R}'$), then $t \xrightarrow{*}_{\mathcal{R}} v$ if and only if $t \xrightarrow{*}_{\mathcal{R}'} v$*

The f-sequentialization of a TRS is an interesting transformation because repeated applications of it lead to an overlapping inductively sequential TRS. We formalize this claim below.

LEMMA 1. *Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a left-linear constructor-based TRS, f a defined operation of $\mathcal{R}$ and $\mathcal{R}'$ the f-sequentialization of $\mathcal{R}$.*

1. *Operation f in $\mathcal{R}'$ is overlapping inductively sequential*
2. *Every operation in $\Sigma' - \Sigma$ is overlapping inductively sequential*

Definition 3. [Sequentialization] Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a left-linear constructor-based TRS and $f_1, f_2, \ldots f_n$ be an enumeration of all the operations in Σ that are not overlapping inductively sequential. Let $\mathcal{R}_0$ denote $\mathcal{R}$ and $\mathcal{R}_i$ denote the f_i-sequentialization of $\mathcal{R}_{i-1}$ for $i = 1, \ldots, n$. TRS $\mathcal{R}_n$ is called the *sequentialization* of $\mathcal{R}$.

COROLLARY 1. *The sequentialization of a left-linear constructor-based TRS is overlapping inductively sequential.*

4.4 Assessment

Given a constructor-based possibly conditional TRS, $\mathcal{R}$, we have shown how to construct an overlapping inductively sequential TRS, $\mathcal{R}'$, which defines the same computations of $\mathcal{R}$. As far as functional logic computations are concerned, TRS $\mathcal{R}'$ is preferable to $\mathcal{R}$ because we know a strategy for $\mathcal{R}'$, *INS* [3], which has been proven sound, complete and theoretically very efficient. Strategy *INS* has two noteworthy properties:

1. Modulo non-deterministic choices, *INS* executes only needed steps.

2. Modulo non-deterministic choices, *INS* is sequential.

Property (1) implies that computations do not execute useless steps. Property (2) suggests that implementations of *INS* should be simpler and more efficient than those of strategies without this property. The reason is that to achieve operational completeness, a *don't know* non-deterministic choice requires splitting a computation into parallel threads. However, each thread is *sequential*. Sequentiality is an elusive property [7]. Loosely speaking, it means that given any term t, we determine the next step of the evaluation of t "*by looking at* t *only*." Therefore, each thread of computation originating from a non-deterministic choice is independent of any other thread. This independence implies that little or no control between threads is required and simplifies the implementation. Both properties make *INS* an ideal strategy for functional logic computations and a practical one, too. In fact, a recent implementation of *INS* [6] is conceptually simple, compact, efficient, *operationally* complete, and easy to integrate with residuation as described in [13].

The final aspect of our work to consider is the possible loss of efficiency and/or laziness incurred by the transformation of Sect. 4. The deconditionalization does not add any overhead. In fact, the proof of point (4) of Theorem 1 is based on the fact that the condition of a rewrite rule is equally evaluated by a TRS and its deconditionalization. We see no reason why the implementations of the two systems should differ in this aspect.

The sequentialization of a TRS, according to Definition 2, may contain more defined operations and execute more steps for the same computation. The additional operations increase the size of a program, but pose no execution overhead. The number of additional steps can be precisely measured. Referring to the notation of Definition 2, each step in $\mathcal{R}$ at the position of a term rooted by a non overlapping inductively sequential operation requires two steps in $\mathcal{R}'$. The first of these two steps can be entirely eliminated by an optimization that moves the non-deterministic choice up to the caller. The following fragment of code shows this elimination on an example.

Example 6. Continuing Example 5, operation `insert` is called by operation `permute`, also found in [11], as follows:

```
permute([]) → []
permute([X|Xs]) → insert(X,permute(Xs))
```

If we replace the second rule of `permute` with the following one, the additional step introduced by the sequentialization of the original TRS is eliminated.

```
permute([X|Xs]) → insert₁(X,permute(Xs))
                | insert₂(X,permute(Xs))
```

This transformation is always possible, since in the f-sequentialization of a TRS any f-rooted term is a redex.

The above code can be further optimized. Every $\texttt{insert}_1$-rooted term is reducible. These reductions can be executed at compile time, e.g., in the rewrite rules of a program. The consequence is that operation $\texttt{insert}_1$ can be entirely eliminated as well. Furthermore, during the f-sequentialization of a TRS, *useless rules* [2, Def. 17], such as that shown in Example 3, are easily discarded. Thus, it is possible that some computations in the sequentialization of a TRS are shorter than in the original TRS.

Non overlapping inductively sequential operations are less structured than overlapping inductively sequential ones. Going back to our introductory analogy, they are analogous to *goto* statements in an imperative program. The number of these operations should be small in most programs and the number of steps involving these operations should be small in most computations. Thus, even without any optimization, the overhead introduced by sequentializing a TRS should be small in most cases.

There is, though, one very specific situation in which sequentializing a TRS may be detrimental. Rewriting computations in weakly orthogonal TRSs may require a non-deterministic choice of which subterm of term should be evaluated. By contrast to overlapping inductively sequential TRSs, these choices do not require splitting a computation into parallel threads, which is an expensive task, because in weakly orthogonal TRSs critical pairs are trivial. Instead, all the subterms of these non-deterministic choices are evaluated in parallel within a *single* thread of computation. Our transformation constructs a TRS in which each subterm is evaluated in an independent parallel thread of computation. Note that both CLNC and the demand driven strategy presented in [17] have the same behavior; thus our approach is still competitive even in this situation.

Parallel narrowing [4] is a strategy that behaves as efficiently as theoretically possible in the situation we are discussing. It would be feasible to include the relevant aspects of parallel narrowing in our approach, but it remains to be proven that any theoretical advantage would be preserved in practice. Parallel narrowing is fairly more complicated than *INS*, therefore its implementations are fairly less efficient. More important, it is possible to evaluate several subterms in parallel in a single thread of computation only for *rewriting* steps, not for general *narrowing* steps, i.e., when variables are instantiated.

4.5 Extra variables

If $l \to r$ is a rewrite rule, it is usual to require that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. This condition is unnecessary in our framework. A variable in r that does not occur in l is called an *extra* variable. We allow unrestricted extra variables in rewrite rules. An unconditional rewriting or narrowing step should treat an extra variable as a constant, i.e., the step should not instantiate it. A conditional step may instantiate an extra variable if the variable occurs also in a condition. Let x be an extra variable of a rewrite rule, $l \to r \Leftarrow c$. When this rule is fired on a term t, yielding t', either variable x or some instance of x containing fresh variables may by *introduced* in t'. Since t' itself, rather than t, could be the initial term of a computation, neither x nor other fresh variables in an instance of x create any significant conceptual problem.

5. CONCLUSION

We have presented an approach to narrowing computations in left-linear constructor-based conditional TRSs. This class is the largest ever proposed for functional logic programming. Our approach transforms a TRS in this class into a target TRS belonging to a class for which there exists a sound, complete and relatively efficient narrowing strategy. Our transformation adds little, if any, overhead to most computations. Computations in the target systems can be implemented with relative ease and efficiency. Therefore, in addition to ensuring the soundness and completeness of computations, a non-negligible advantage of our approach is its practicality. Our results support and complement previous efforts aimed at using left-linear constructor based conditional TRSs for modeling functional logic programs and aimed at using narrowing for implementing functional logic computations.

Acknowledgement

I am grateful to Eva Ullan and the anonymous reviewers for their comments and suggestions on the initial version of this paper.

6. 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 3rd International Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.

[3] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.

[4] S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 138–152. MIT Press, 1997.

[5] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, July 2000.

[6] S. Antoy, M. Hanus, B. Massey, and F. Steiner. An implementation of narrowing. In *Principles and Practice of Declarative Programming (PPDP'01)*, Sept. 2001. (In this volume).

[7] S. Antoy and A. Middeldorp. A sequential strategy. *Theoretical Computer Science*, 165:75–95, 1996.

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

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

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

[11] J. C. González Moreno, F. J. López Fraguas, M. T. Hortalá González, and M. Rodríguez Artalejo. An approach to declarative programming based on a rewriting logic. *The Journal of Logic Programming*, 40:47–87, 1999.

[12] M. Hanus. On extra variables in (equational) logic programming. In *Proc. Twelfth International Conference on Logic Programming*, pages 665–679. MIT Press, 1995.

[13] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.

[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] G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*. MIT Press, Cambridge, MA, 1991.

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

[17] 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.

[18] M. Marchiori. Unravelings and ultra-properties. In *Proceedings of the Fifth International Conference on Algebraic and Logic Programming (ALP'96)*, pages 107–121. Springer LNCS 1139, 1996.

[19] A. Middeldorp and S. Okui. A deterministic lazy narrowing calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

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

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

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

Appendix

In this appendix, we informally discuss the proof of a non-trivial result of this paper. Most of our claims are relatively straightforward to prove except for point 4 of Theorem 1. Although this statement is somewhat intuitive, proving the "if" part requires a considerable infrastructure.

A fundamental result for reasoning about rewrite derivations is the *Parallel Moves Lemma* and the consequent notions of *descendant* [15] and *trace* [9] of a term. We cannot apply these concepts to our discussion since, in general, they do not hold for non-orthogonal TRSs. The two crucial conditions, satisfied by orthogonal TRSs, which ensure the Parallel Moves Lemma are left-linearity and non-ambiguity. The left-linear constructor-based TRSs obviously satisfy the first condition, but not the second. However, as we remarked in Section 4.3, the constructor discipline severely curtails ambiguity. The left-hand sides of two rewrite rules can overlap only at the root. Therefore, it is possible to formulate, even for this large class, a weaker notion of the Parallel Moves Lemma and consequently those of descendant and trace of a term. The key of this formulation is to restrict the moves to "compatible" steps, as we did in [3] for narrowing.

Two steps of a same term, say $t \to_{p_1,R_1} u_1$ and $t \to_{p_2,R_2} u_2$, are *compatible* if $p_1 = p_2$ implies $R_1 = R_2$. In particular, steps at distinct positions are always compatible, whereas steps at a same position are compatible only if they use the same rewrite rule. Indeed, we could further generalize this condition for weak ambiguity, but this generalization is not necessary for our proof.

Coming to the proof of point 4 of Theorem 1, let $\mathcal{R}$ be a left-linear constructor-based TRS with equality and $\mathcal{R}'$ its deconditionalization. Furthermore, let t and v be a term and a value in $\mathcal{R}$ such that $A : t \xrightarrow{*}_{\mathcal{R}'} v$. The proof is by induction on a *cost* function, denoted by α, that takes a computation and returns a non-negative integer.

Function α tallies the "cost" in a derivation of applying rewrite rules with the right-hand side rooted by the `if` symbol, i.e., deconditionalized rewrite rules. Intuitively, the application of a deconditionalized rewrite rule should have unit cost; hence the proof would be an induction of the number of steps of a derivation that fire a deconditionalized rewrite rule. Unfortunately, a complication arises in a special case. Initially, to ease understanding, we ignore this complication and begin with this simple formulation of α. Later, we explain the complication and how it is solved by a refinement of the cost function.

Let $A : t \xrightarrow{*} v$ a derivation in $\mathcal{R}'$ and let $A[i]$ denote the suffix of A starting with the i-th term. We define $\alpha(A[i]) = \delta + \alpha(A[i+1])$, where $\delta = 1$ if the rewrite rule applied to the i-th term of A is deconditionalized, and $\delta = 0$ otherwise. As we said earlier, this definition of α simply counts the number of steps of A that apply a deconditionalized rewrite rule and the proof by induction on this number.

The base case is trivial, since each step of A can be executed in $\mathcal{R}$ as well. For the induction case, without loss of generality, suppose that $t \to_{p,R} u \xrightarrow{*} v$, where $R = l \to \mathtt{if}(c, r)$. By definition, $u = t[\mathtt{if}(c, r)\sigma]_p$ for some substitution σ. The potential problem of this situation is that if $c\sigma$ cannot be evaluated to `success`, we cannot apply the rewrite rule $l \to r \Leftarrow c$ in $\mathcal{R}$ and thus prove that $t \to t[r]_p \xrightarrow{*}_{\mathcal{R}} v$. However, there is a solution.

Consider a derivation, B, in $\mathcal{R}'$ that executes the same steps of A except that every step at a position that originates within $t|_p$, i.e., it is a descendant or a trace of a term at or below p in t, is omitted. Derivation B is formalized with notions that depend on the generalization of the Parallel Moves Lemma that we sketched earlier. An intuitive formulation is to consider $t[\blacksquare]_p$, where $\blacksquare$ is a new irreducible symbol, and to execute every step of A that can still be executed in B. Derivation B is finite and there are only two possible outcomes for its last term. If the last term of B is v, then the first step of A can be avoided and by the induction hypothesis $t \xrightarrow{+}_{\mathcal{R}} v$. If the last term of B is not v, then there exists a step of A that is possible because a term originating from $t|_p$ is constructor-rooted, whereas the corresponding step of B is not possible because of the presence of $\blacksquare$. This implies that in A, term $\mathtt{if}(c, r)\sigma$ is evaluated to a constructor-rooted term, and therefore $c\sigma$ is evaluated to `success`. Thus, there exists a derivation $C : t \xrightarrow{+}_{\mathcal{R}'} t' \xrightarrow{*}_{\mathcal{R}'} v$, where t' contains only operations in the signature of $\mathcal{R}$. If the cost of the suffix of C starting with t' were smaller than $\alpha(A)$, we could apply the induction hypothesis in this case, too, and conclude that $t \xrightarrow{+}_{\mathcal{R}} v$. Unfortunately, this is not always the case if α counts the number of applications of deconditionalized rewrite rules in a derivation.

Before we explain this point, observe that the failure of B to reduce t to v does not imply that $t|_p$ is needed, since there is no notion of needed redex for the left-linear constructor-based TRSs. It only implies that $t_{|p}$ must be contracted within the context of the steps of A to reach v.

The cost of the suffix of C starting with t' is not always smaller than $\alpha(A)$, because the order in which the redexes of a term, say t, are contracted affects the total number of steps of a derivation of t to a value. Contracting outer redexes may multiply the occurrences of inner redexes and thus lengthen a derivation. Going back to the proof sketch presented earlier, it could happen that $\alpha(A) < \alpha(C)$ and thus it would be wrong to apply the induction hypothesis. To correct the problem, it suffices to refine the function α.

Since the problem stems from multiple occurrences of descendants of a same redex, a classic approach is to bundle together in a *family* all these descendants. A multistep contracting one or more members of a same family is assigned unit cost regardless of the size of the family. We have formalized this concept in [5, Def. 17] for narrowing steps in inductively sequential TRSs. In the present situation, we are only concerned with rewriting steps, but the concept is slightly more complicated by the possibility that two redexes with a same ancestor may have different contractums. To solve this problem, we call *mutual clones* the redexes of a derivation that both have a same ancestor; hence, are in the same family and have a same contractum. We assign unit cost to a multistep that contracts a non-empty set of mutual clones, i.e., increasing the number of mutual clones contracted in a step does not increase the cost of the step. This is a sensible decision even in practice because implementations of narrowing may (should) share mutual clones in the representation of a term.

The formal definition of this more sophisticated cost function is somewhat laborious, but conceptually no more complicated than that in [5, Def. 17]. This definition fixes the defect of our initial approximation of the induction case of the proof sketch presented earlier since multiple occurrences of descendants of a same redex no longer increase the cost of a suffix of a derivation.

Note that the call-time semantics implies that all members of a same family are mutual clones—a condition that would somewhat simplify the discussion. Our treatment, though, is not limited by the semantic option that may be adopted for a functional logic language or program. Both the overall structure of our proof and our definition of cost accommodate the need-time semantics, too.