Overlapping Rules and Logic Variables in Functional Logic Programs*

October 12, 2005

Sergio Antoy[1] Michael Hanus[2]

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

[2] Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.
mh@informatik.uni-kiel.de

Abstract. Functional logic languages extend purely functional languages with two features: operations defined by overlapping rules and logic variables in both defining rules and expressions to evaluate. In this paper, we show that only one of these features is sufficient in a core language. On the one hand, overlapping rules can be eliminated by introducing logic variables in rules. On the other hand, logic variables can be eliminated by introducing operations defined by overlapping rules. The proposed transformations between different classes of programs not only give a better understanding of the features of functional logic programs but also are useful to simplify implementations of functional logic languages.

1 Motivation

Functional logic languages [18] integrate the best features of functional and logic languages in order to provide a variety of programming concepts. For instance, the concepts of demand-driven evaluation and higher-order functions from functional programming can be combined with logic programming features like computing with partial information (logic variables), constraint solving, and nondeterministic search for solutions. In contrast to purely functional languages, functional logic languages allow computations with overlapping rules (i.e., more than one rule can be applied to evaluate a function call) and logic variables (i.e., unbound variables occurring in the initial expression and/or rules, also called extra variables). Operationally, these features are supported by nondeterministic computation steps.

Functional logic languages are modeled by constructor-based term rewriting systems (TRS) with narrowing as the evaluation mechanism. A crucial choice in the design of a language, both at the source level and the implementation level, is the class of rewrite systems used to model the programs. Early languages (e.g., Babel [26] and K-Leaf [17]) were modeled by weakly orthogonal, constructor-based TRSs. Larger classes provide more expressiveness. Thus, modern languages, such as Curry [19, 21] and $\mathcal{TOY}$ [24], are modeled by the whole class of the constructor-based rewrite systems with extra variables. However, the implementation of a language modeled by a smaller class is likely to be simpler and/or more efficient.

* This work was partially supported by the German Research Council (DFG) grant Ha 2457/5-1 and the NSF grant CCR-0218224.

For the above reason, program transformation among different classes of TRSs is an interesting research subject. The goal is to transform a program in the source language into an equivalent program in a language, referred to as the *core* language, that is conceptually simpler or could be implemented more efficiently. For example, [5] shows that any conditional constructor-based TRS can be transformed into an unconditional overlapping inductively sequential TRS [4]. The target class is a proper subclass of the source class, a situation that leads to conceptual and practial benefits. This paper studies two transformations similar to that described in [5] and with the same intent.

The first transformation maps the overlapping inductively sequential TRS with or without extra variables into the inductively sequential TRS with extra variables. This shows that if a language allows extra variables, then, at the core level, overlapping is not necessary. Of course, at the source level overlapping is a feature that contributes to the expressiveness of a language and therefore is desirable.

The second transformation eliminates logic variables from computations within the overlapping inductively sequential TRS. By "logic variables" we mean extra variables in rewrite rules and variables, which are free or unbound, in expressions to evaluate. A somewhat unexpected, though immediate, consequence of this transformation is that the power of narrowing computations can be obtained by mere rewriting. As for the previous transformation, at the source level logic variables contribute to the expressiveness of a language and therefore are desirable.

Loosely speaking, these results can be understood as the possibility to trade in a core language logic variables for a rather disciplined form of rule overlapping and vice versa. Section 2 reviews concepts and notations used in this paper. Section 3 defines the transformation that replaces overlapping with extra variables and proves its correctness. Section 4 defines the transformation that replaces logic variables with overlapping and proves its correctness. Section 5 offers our conclusion.

2 Preliminaries

In this section we review some term rewriting [10, 16] notations and functional logic programming [18] concepts used in the remaining of this paper.

We consider a many-sorted *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. We write $c/n \in \mathcal{C}$ and $f/n \in \mathcal{F}$ for n-ary constructor and operation symbols, respectively. Given a set of sorted variables $\mathcal{X}$, the set of well-sorted *terms* and *constructor terms* are denoted by $\mathcal{T}(\Sigma, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. We write $\mathcal{V}ar(t)$ for the set of all the variables occurring in a term t. A term t is *ground* if $\mathcal{V}ar(t) = \varnothing$. A term is *linear* if it does not contain multiple occurrences of a variable. A term is *operation-rooted* (*constructor-rooted*) if its root symbol is an operation (constructor). A *head normal form* is a term that is not operation-rooted, i.e., it is a variable or a constructor-rooted term. We write $\overline{o_k}$ for a sequence of objects $o_1, \ldots, o_k$.

Example 1. In the following, we write datatype declarations in Curry syntax [21], i.e., a sort S is defined by enumerating its constructors in the form

data S = C_1 s_{11} $\ldots$ s_{1a_1} | $\ldots$ | C_n s_{n1} $\ldots$ s_{na_n}

Thus, C_i is a constructor of sort S and arity a_i with argument sorts $s_{i1}, \ldots, s_{ia_i}$. For instance, the sorts of Boolean values and natural numbers in Peano's notation are defined as

```
data Bool = True | False
data Nat  = O | S Nat
```

$\square$

A *pattern* is a linear term of the form $f(t_1, \ldots, t_n)$ where $f/n \in \mathcal{F}$ is an operation symbol and $t_1, \ldots, t_n$ are constructor terms. A constructor-based rewrite system is a set of pairs of terms or *rewrite rules* of the form

$$l \to r$$

where l is a pattern and l and r are of the same sort. An operation f is *defined* by all the rewrite rules whose left-hand side is rooted by f. A *functional logic program* is a constructor-based rewrite system. Traditionally, term rewriting systems have the additional requirement $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. However, in functional logic programming variables occurring in $\mathcal{V}ar(r)$ but not in $\mathcal{V}ar(l)$), called *extra variables*, are often useful. Therefore, we allow rewrite rules with extra variables in functional logic programs. We denote the set of extra variables of a rewrite rule $l \to r$, defined as $\mathcal{V}ar(r) \backslash \mathcal{V}ar(l)$, with $\mathcal{E}var(l \to r)$.

To formally define computations w.r.t. a given program, additional notions are necessary. A *position* p in a term t is represented by a sequence of natural numbers. Positions are used to identify specific subterms. Thus, $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ with the term s (see [16] for details). A *substitution* is an idempotent mapping $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$ such that its *domain* $\mathcal{D}om(\sigma) = \{x \mid \sigma(x) \neq x\}$ is finite and x and $\sigma(x)$ are of the same sort for all variables x. We denote a substitution σ by the finite set $\{x \mapsto \sigma(x) \mid x \in \mathcal{D}om(\sigma)\}$. In particular, $\varnothing$ denotes the identity substitution. We denote by $\sigma|_V$ the restriction of a substitution σ to a set of variables V. A *(ground) constructor substitution* σ has the property that $\sigma(x)$ is a (ground) constructor term for all $x \in \mathcal{D}om(\sigma)$. The composition $\sigma \circ \eta$ of two substitutions is defined by $(\sigma \circ \eta)(x) = \eta(\sigma(x))$ for all variables x. Substitutions are extended to morphisms on terms in the obvious way. The *subsumption ordering* is a binary relation on terms defined by $u \leq v$ if there is a substitution σ with $\sigma(u) = v$. In this case, v is also called an *instance* of u. If, in addition, v is a (ground) constructor term, we call it *(ground) constructor instance.* If $u \leq v$ and $v \leq u$, then u and v differ only for a renaming of variables. We write $u < v$ if $u \leq v$ and $v \not\leq u$. A *unifier* of two terms s and t is a substitution σ such that $\sigma(s) = \sigma(t)$. The unifier σ is *most general* if for any other unifier σ' there exists a substitution η with $\sigma' = \sigma \circ \eta$. Furthermore, we denote by $s \lhd t$ the most general unifier of s and t restricted to $\mathcal{V}ar(s)$.

A *rewrite step* $t \to_{p,l\to r,\eta} t'$ w.r.t. a given rewrite system $\mathcal{R}$ is defined if there are a position p in t, a rule $l \to r \in \mathcal{R}$ with fresh variables, and a substitution η with $t|_p = \eta(l)$ such that $t' = t[\eta(r)]_p$. We impose the condition on the freshness of the variables since we allow extra variables in rewrite rules. The indices in the notation of a rewrite step are omitted when inconsequential. A term t is called *irreducible* or in *normal form* if there is no term s such that $t \to s$. $\overset{+}{\to}$ and $\overset{*}{\to}$ denote the transitive and reflexive-transitive closure of the relation $\to$, respectively.

Functional logic languages compute solutions of free variables occurring in expressions by instantiating these variables to constructor terms so that a rewrite step becomes applicable. The combination of variable instantiation and rewriting is called *narrowing*. Formally, $t \rightsquigarrow_\sigma t'$ is a *narrowing step* if $\sigma(t) \rightarrow_{p,l\rightarrow r,\eta} t'$ where σ is a substitution, $t|_p$ is not a variable, and $\mathcal{D}om(\eta) \subseteq \mathcal{V}ar(l)$. We denote by $t_0 \overset{*}{\rightsquigarrow}_\sigma t_n$ a sequence of narrowing steps $t_0 \rightsquigarrow_{\sigma_1} \ldots \rightsquigarrow_{\sigma_n} t_n$ with $\sigma = \sigma_1 \circ \cdots \circ \sigma_n$ (if $n = 0$ then $\sigma = \varnothing$). We omit the substitution in the notation of both narrowing steps and sequences when irrelevant to the discussion.

The requirement that $\mathcal{D}om(\eta) \subseteq \mathcal{V}ar(l)$, as in [5], ensures that no extra variable in a rule is instantiated during a narrowing step. An extra variable in a rewrite rule is generally intended as a place holder for any term, e.g., see [11] where extra variables are allowed in the conditions of rewrite rules. In constructor-based rewrite systems, a more suitable convention should allow an extra variable to stand only for constructor terms, since terms that cannot be reduced to a constructor term are intended as errors. By contrast, requiring that extra variables remain uninstantiated in a rewrite step appears as treating extra variables as constants, thus foregoing the expressive power that they provide. However, when computations are performed by narrowing, particularly using an efficient strategy, it seems most sensible to avoid instantiating extra variables in the step that introduces them. The reason is that these variables become logic variables in subsequent steps and therefore may be narrowed. The advantage of instantiating them in a narrowing step after they are introduced, as opposed to instantiating them in the step that introduces them, is that the latter would have no information on choosing useful instantiations, whereas the former could instantiate them with choices useful to perform a step. In particular, efficient strategies such as [4, 7] will instantiate logic variables only as far as necessary to perform needed steps. This level of specialization seems impossible to achieve at the time extra variables are introduced, unless the step introducing them performs some kind of lookahead.

The general definition of narrowing allows too many narrowing steps in a term so that it is not useful in practice. Therefore, older narrowing strategies (see [18] for a detailed account), influenced by the resolution principle, require that the substitution used in a narrowing step is a most general unifier of the term being replaced and the left-hand side of the applied rule. As shown in [7], this condition prevents the development of optimal evaluation strategies. Therefore, more recent narrowing strategies relax this requirement but provide other constructive methods to compute a small set of unifiers and positions used in narrowing steps [6]. In particular, *needed* or *demand-driven* strategies perform narrowing steps only if they are necessary to compute a result. Such strategies are defined on more restricted classes of rewrite systems that will be defined next.

An important narrowing strategy, needed narrowing [7], is defined on the subclass of the *inductively sequential* TRSs. This class can be characterized by definitional trees [3] that are also useful to formalize and implement demand-driven narrowing strategies. Since only the left-hand sides of rules are important for the applicability of needed narrowing, the following formulation of definitional trees [4] considers patterns partially ordered by subsumption.

A *definitional tree* of an operation f is a non-empty set T of linear patterns partially ordered by subsumption having the following properties:

Leaves property: The maximal elements of T, called the *leaves*, are exactly the (variants of) the left-hand sides of the rules defining f. Non-maximal elements are also called *branches*.

Root property: T has a minimum element, called the *root*, of the form $f(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are pairwise distinct variables.

Parent property: If $\pi \in T$ is a pattern different from the root, there exists a unique $\pi' \in T$, called the *parent* of π (and π is called a *child* of π'), such that $\pi' < \pi$ and there is no other pattern $\pi'' \in \mathcal{T}(\Sigma, \mathcal{X})$ with $\pi' < \pi'' < \pi$.

Induction property: All the children of a pattern π differ from each other only at a common position, called the *inductive position*, which is the position of a variable in π.[1]

An operation is called inductively sequential if it has a definitional tree. Traditionally, it is also required that the rules do not contain extra variables [7]. Here, we relax this requirement: A TRS is *inductively sequential with extra variables* (*ISX*) if all its defined operations are inductively sequential. Intuitively, inductively sequential functions are defined by structural induction on the argument types. Purely functional programs and the vast majority of functions in functional logic programs are inductively sequential.

Example 2. The following operations are inductively sequential w.r.t. the datatype declarations of Example 1:

```
leq(0,x)         → True
leq(S(x),0)      → False
leq(S(x),S(y))  → leq(x,y)

cond(True,x) → x

nine → S(S(S(S(S(S(S(S(S(0)))))))))
```

The operation `smallnum` denotes a number less than ten and is defined by an ISX rule containing an extra variable `x`:

```
smallnum → cond(leq(x,nine),x)
```
□

Functional logic languages extend purely functional languages by allowing overlapping rules. We are interested only in a disciplined form of overlapping. Two distinct rewrite rules $l_1 \to r_1$ and $l_2 \to r_2$ are called *overlapping* if the left-hand sides l_1 and l_2 are variants of each other, i.e., they are equal by subsumption. We denote the set of all rules with the same left-hand side l by the single (meta) rule $l \to r_1 \,?\, \cdots \,?\, r_k$, where "?" is a meta symbol and $r_1, \ldots, r_k$ are the right-hand sides. A TRS is *overlapping inductively sequential* (*OIS*) if all its defined operations are inductively sequential when overlapping rules with identical left-hand sides are joined into a single rule as above. The purpose of this paper is to show that an ISX program executed by narrowing can be transformed into an OIS program executed by rewriting and vice versa, i.e., the classes ISX and OIS loosely speaking have the same expressiveness.

Next, we define the needed narrowing strategy on inductively sequential rewrite systems.

Definition 1. Let $\mathcal{R}$ be an inductively sequential TRS where each function symbol has a uniquely associated definitional tree. We define the function λ from

[1] There might exist distinct definitional trees of an operation. In this case one can use any tree for computing a needed narrowing step of a term since the need of the step does not depend on the selected tree.

operation-rooted terms to sets of triples (position, rule, substitution) as follows. Let $t = f(t_1, \ldots, t_n)$ be an operation-rooted term, T the definitional tree associated to f, and π a maximal pattern of T that unifies with t. Then $\lambda(t)$ is the least set satisfying

$$\lambda(t) \ni \begin{cases} (\Lambda, \pi \to r, t \lhd \pi) & \text{if } \pi \text{ is a leaf of } T \text{ and } \pi \to r \\ & \quad \text{is a variant of a rewrite rule} \\ (q \cdot p, R, \eta \circ \sigma) & \text{if } \pi \text{ is a branch of } T, \\ & \quad \text{where } q \text{ is the inductive position of } \pi, \\ & \quad \eta = t \lhd \pi, \text{ and } (p, R, \sigma) \in \lambda(\eta(t|_q)) \end{cases} \qquad \square$$

In each recursive step during the computation of λ, a position and a substitution is composed with the results computed by the recursive call. Thus, each needed narrowing step can be represented as $(p_1 \cdots p_k, R, \sigma_1 \circ \cdots \circ \sigma_k)$, where $p_k = \Lambda$, p_j is an inductive position for all $j \in \{1, \ldots, k-1\}$, and σ_j a most general unifier restricted to the term variables computed in each recursive call for all $j \in \{1, \ldots, k\}$. This representation of a needed narrowing step is called its *canonical decomposition.*

Proposition 1 ([4]). *Let $\mathcal{R}$ be an overlapping inductively sequential TRS and t an operation-rooted term. If $(p, l \to r, \sigma) \in \lambda(t)$, then $t \leadsto_{p,l\to r,\sigma} \sigma(t[r]_p)$ is a* needed *narrowing step, also denoted by $t \overset{NN}{\leadsto}_{p,l\to r,\sigma} \sigma(t[r]_p)$.*

The need of the step computed by λ in Proposition 1 is *modulo the nondeterministic choice* of the right-hand side. The term t cannot be narrowed to a constructor term without a step at p with a rule $l \to r'$. However, it may be possible that $r \neq r'$.

3 Eliminating Overlapping Rules

In this section we show that using rules with multiple right-hand sides does not increase the expressiveness of a functional logic language already providing inductively sequential rewrite systems with extra variables. For this purpose, we introduce a transformation from OIS into ISX systems and prove that needed narrowing computes the same results on the original and the transformed system.

Definition 2 (Transformation from OIS into ISX). We define a transformation *OE* (*Overlapping Elimination*) on TRSs. Non-overlapping rewrite rules are not changed. Overlapping rewrite rules of the form $f(\overline{t_n}) \to r_1 \,?\, \cdots \,?\, r_k$ are replaced by a single rule $f(\overline{t_n}) \to f'(y, \overline{x_l})$ where $\mathcal{V}ar(\overline{t_n}) = \{x_1, \ldots, x_l\}$, y is a new free variable, and f' is a new function symbol defined by the new rules

$$\begin{array}{l} f'(I_1, \overline{x_l}) \to r_1 \\ \vdots \\ f'(I_k, \overline{x_l}) \to r_k \end{array}$$

The constants I_j are the elements of a new index type defined by

data Ix = I_1 | $\cdots$ | I_k

In practice, one can use the same index type (e.g., natural numbers) for all the rules. $\square$

The transformation only adds new function and constructor symbols. Thus, every term w.r.t. the original signature is also a term w.r.t. the transformed signature. In the following, we denote the original TRS by $\mathcal{R}$ and the transformed TRS by $\mathcal{R}' = OE(\mathcal{R})$.

Example 3. The following operation `coin` epitomizes an overlapping inductively sequential operation:

```
coin → 0 ? 1
```

The *OE* transformed program is:

```
data Icoin = I0 | I1
coin → coin'(y)
coin'(I0) → 0
coin'(I1) → 1
```
□

Example 4. Consider an operation `parent` that non-deterministically returns either the mother or the father of the argument:

```
parent(x) → mother(x) ? father(x)
```

The *OE* transformed program is:

```
data Iparent = I0 | I1
parent(x) → parent'(y,x)
parent'(I0,x) → mother(x)
parent'(I1,x) → father(x)
```
□

Proposition 2. *If $\mathcal{R}$ is overlapping inductively sequential, then the transformed system $\mathcal{R}'$ is inductively sequential with extra variables.*

Proof. Since the left-hand sides of the rules in $\mathcal{R}$ are not changed and the rules of the new function symbols are obviously inductively sequential, $\mathcal{R}'$ is inductively sequential. Furthermore, since all overlapping rules in $\mathcal{R}$ are eliminated and the rules of the new function symbols are not overlapping, the proposition holds. □

To claim the correctness of the transformation, we need to show that every computation in the original system has a corresponding computation in the transformed system and vice versa. For this purpose, we need two auxiliary results. The following lemma shows that any narrowing step in the original system can be simulated in the transformed system by either the same step or two consecutive steps using the introduced rules.

Lemma 1 (Completeness of *OE*). *Let $t \in \mathcal{T}(\Sigma, \mathcal{X})$. If $t \rightsquigarrow_{p,R,\sigma} t'$ with $l \to r \in \mathcal{R}$, then $t \overset{+}{\rightsquigarrow}_{\sigma'} t'$ w.r.t. $\mathcal{R}'$ with $\sigma =_{\mathcal{V}ar(t)} \sigma'$.*

Proof. If R is a rule in $\mathcal{R} \cap \mathcal{R}'$, the lemma holds immediately. Otherwise, there are overlapping rules $l \to r_1 \,?\, \cdots \,?\, r_k$ in $\mathcal{R}$ and $R = f(t_1, \ldots, t_n) \to r_j$ for some j. Since $t \rightsquigarrow_{p,R,\sigma} t'$, by definition of narrowing, $t|_p = \sigma(f(t_1, \ldots, t_k))$ and $t' = \sigma(t)[\sigma(r_j)]_p$ for some j. Then

$$t \rightsquigarrow_{p,f(\overline{t_n}) \to f'(y,\overline{x_l}),\sigma} \sigma(t)[f'(y, \overline{\sigma(x_l)})]_p \rightsquigarrow_{p,f'(I_j,\overline{x_l}) \to r_j,\{y \mapsto I_j\}} \sigma(t)[\sigma(r_j)]_p$$

is a narrowing derivation w.r.t. $\mathcal{R}'$ with a fresh variable y. Since $\sigma' = \{y \mapsto I_j\} \circ \sigma$ and $y \notin \mathcal{V}ar(t)$, $\sigma =_{\mathcal{V}ar(t)} \sigma'$. □

The next lemma shows that every needed narrowing step in the transformed system that introduces a function symbol not occurring in the signature of the original system is immediately followed by a needed narrowing step that removes this symbol.

Lemma 2 (Soundness of *OE*). *Let $f(\overline{t_n}) \rightarrow r_1 \,?\, \cdots \,?\, r_k$ be overlapping rules of an OIS TRS and $R = f(\overline{t_n}) \rightarrow f'(y, \overline{x_l})$ and $R_j = f'(I_j, \overline{x_l}) \rightarrow r_j$, for $j \in \{1, \ldots, k\}$, be the corresponding rules introduced by the OE transformation. If $t \in \mathcal{T}(\Sigma, \mathcal{X})$ and $t \overset{\text{NN}}{\rightsquigarrow}_{p,R,\sigma} t'$, then $t' \overset{\text{NN}}{\rightsquigarrow}_{p,R_j,\{y \mapsto I_j\}} t'[\sigma(r_j)]_p$ are the only needed narrowing steps applicable to t'.*

Proof. Let $(p_1 \cdots p_k, R, \sigma_1 \circ \cdots \circ \sigma_k)$ be the canonical decomposition of the first needed narrowing step. Let π_m, for $m \in \{1, \ldots, k-1\}$, be the branch of the definitional tree used in conjunction with position $p_1 \cdots p_m$ in the computation of $\lambda(t)$. Observe that π_1 is used at Λ, π_2 is used at p_1, etc. Let Q be the set of positions in π_m. Since the replacement in the narrowing step of t is at a position below $p_1 \cdots p_{m-1}$, for each position $q \in Q$, $(\sigma_1 \circ \cdots \circ \sigma_m(t|_{p_1 \cdots p_{m-1}}))|_q = (t'|_{p_1 \cdots p_{m-1}})|_q$. Therefore, π_m is the only maximal pattern in its definitional tree that unifies with $t'|_{p_1 \cdots p_{m-1}}$ and the unifier is the identity on $\mathcal{V}ar(t')$: no pattern greater than π_m unifies with $t'|_{p_1 \cdots p_{m-1}}$, otherwise, π_m would not be maximal in the computation of $\lambda(t)$. Furthermore, $t'|_{p_1 \cdots p_k}$ unifies with any leaf in the definitional tree of f': in the definitional tree of f' the leaves have the form $f'(I_j, \overline{x_l})$ and in $t'|_{p_1 \cdots p_k}$ the root is f' and its first argument is a variable.

Thus, $(p, R_j, \{y \mapsto I_j\})$ belongs to $\lambda(t')$ and the narrowing steps exist as claimed. Furthermore, no other needed narrowing step from t' exists due to the uniqueness of π_m for all $m \in \{1, \ldots, k-1\}$. □

The following theorem states the main result of this section—the correctness of the *OE* transformation w.r.t. needed narrowing.

Theorem 1 (Correctness of *OE*). *Let $\mathcal{R}$ be a OIS TRS, $\mathcal{R}' = OE(\mathcal{R})$, and t, s terms of $\mathcal{R}$. The following claims hold.*

Soundness *If $t \overset{\text{NN}}{\rightsquigarrow}^*_{\sigma'} s$ w.r.t. $\mathcal{R}'$, then there exists a derivation $t \overset{\text{NN}}{\rightsquigarrow}^*_{\sigma} s$ w.r.t. $\mathcal{R}$ such that $\sigma =_{\mathcal{V}ar(t)} \sigma'$.*

Completeness *If $t \overset{\text{NN}}{\rightsquigarrow}^*_{\sigma} s$ w.r.t. $\mathcal{R}$, then there exists a derivation $t \overset{\text{NN}}{\rightsquigarrow}^*_{\sigma'} s$ w.r.t. $\mathcal{R}'$ such that $\sigma =_{\mathcal{V}ar(t)} \sigma'$.*

Proof. The completeness is an immediate consequence of Lemma 1. For the soundness, consider a needed narrowing derivation

$$t_0 \overset{\text{NN}}{\rightsquigarrow}_{\sigma'_1} t_1 \overset{\text{NN}}{\rightsquigarrow}_{\sigma'_2} \cdots \overset{\text{NN}}{\rightsquigarrow}_{\sigma'_n} t_n$$

w.r.t. $\mathcal{R}'$ and $t = t_0$, $s = t_n$. We show by induction on n that there exists a corresponding needed narrowing derivation

$$t_0 \overset{\text{NN}}{\rightsquigarrow}^+_{\sigma_1} t_1 \overset{\text{NN}}{\rightsquigarrow}^+_{\sigma_2} \cdots \overset{\text{NN}}{\rightsquigarrow}^+_{\sigma_n} t_n$$

w.r.t. $\mathcal{R}$ and $\sigma'_1 \circ \cdots \circ \sigma'_n =_{\mathcal{V}ar(t)} \sigma_1 \circ \cdots \circ \sigma_n$.

The base case ($n = 0$) vacuously holds. For the induction step, consider the first narrowing step $t_0 \overset{\text{NN}}{\rightsquigarrow}_{\sigma'_1} t_1$. If this step uses a rule from $\mathcal{R} \cap \mathcal{R}'$, the claim follows immediately from the induction hypothesis. Otherwise, there are overlapping rules $R = f(\overline{t_n}) \to r_1 \,?\cdots?\, r_k$ in $\mathcal{R}$ such that $R_1 = f(\overline{t_n}) \to f'(y, \overline{x_l})$ and $R_2 = f'(I_j, \overline{x_l}) \to r_j$, for some $j \in \{1, \dots, k\}$, are the corresponding rules introduced by the *OE* transformation. Since $t_0 \in \mathcal{T}(\Sigma, \mathcal{X})$, by Lemma 2,

$$t_0 \overset{\text{NN}}{\leadsto}_{p_1, R_1, \sigma_1'} t_1 \overset{\text{NN}}{\leadsto}_{p_1, R_2, \sigma_2'} t_2$$

with $\sigma_2' = \{y \mapsto I_j\}$ (where y is a free variable introduced in the first narrowing step). Since R_1 and the overlapping rules R have identical left-hand sides (so that we can assume that their definitional trees are also identical), there exists a needed narrowing step

$$t_0 \overset{\text{NN}}{\leadsto}_{p_1, f(\overline{t_n}) \to r_j, \sigma_1'} t_1[\sigma_1'(r_j)]_{p_1}$$

By Lemma 2, $t_1[\sigma_1'(r_j)]_{p_1} = t_2$ and $\sigma_1' \circ \sigma_2' =_{\mathcal{V}ar(t)} \sigma_1'$. Since $t_2 \in \mathcal{T}(\Sigma, \mathcal{X})$, we can apply the induction hypothesis to the remaining narrowing derivation to prove the claim. □

4 Eliminating Logic Variables

In the previous section, we have shown that the class of the inductively sequential TRSs with extra variables, *ISX*, is at least as expressive as the class of the overlapping inductively sequential TRSs, *OIS*. This result is interesting because it enables us to trade in the implementation of a language the complications of overlapping, or multiple right-hand sides, for the presence of extra variables. Since we already allow extra variables in the *OIS* programs, we simply eliminate overlapping in the transformation.

In this section, we present a somewhat complementary result. We show that the overlapping inductively sequential TRSs, *without extra variables*, denoted *OIS*$^-$, are at least as expressive as the *ISX* programs. We use a transformation that eliminates unbound variables *entirely*, i.e., also from the "top-level" or initial term being evaluated. Therefore, a computation in the OIS$^-$ programs is by rewriting, not narrowing. This result is interesting because it enables us to trade in the implementation of a language the complications of narrowing, in particular the use of substitutions, for the presence of multiple right-hand sides in the program rules.

As for the *OE* transformation, a functional logic program is an overlapping inductively sequential, many sorted, constructor based TRSs with extra variables. This time, though, our goal is to eliminate extra variables, instead of overlappings. Thus, we denote with *XE*, *extra variable elimination*, the new transformation. For any sort S, we consider a constant operation, `instanceOf`S, that enumerates the values of the sort S. We call this operation a *generator* of S.

Definition 3 (`instanceOf`). Let S be a sort defined by a datatype declaration of the form

`data` S `=` $C_1\ t_{11}\ \dots\ t_{1a_1}$ `|` $\dots$ `|` $C_n\ t_{n1}\ \dots\ t_{na_n}$

The operation `instanceOf`S is defined by the overlapping rules

`instanceOf`$S \to C_1$`(instanceOf`t_{11}`,...,instanceOf`t_{1a_1}`)`

```
? ...
? $C_n$(instanceOf$t_{n1}$,...,instanceOf$t_{na_n}$)
```
$\square$

If S is a *primitive* or *builtin* sort, e.g., integers or characters, then we will assume that the operation `instanceOf`S is primitive or builtin as well. However, the following example shows that generators of primitive sorts, even infinite ones, can be coded by ordinary rules.

Example 5. Suppose that a sort “tree of integers” is defined by

```
data TreeInt = Leaf | Branch Int TreeInt TreeInt
```

the generator of `TreeInt` is

```
instanceOfTreeInt
   → Leaf
    ? Branch(instanceOfInt,instanceOfTreeInt,instanceOfTreeInt)
```

Below are two plausible ordinary definitions of the generator of the integers:

```
instanceOfInt = 0 ? genNeg ? genPos
genNeg →  -1 ? genNeg - 1
genPos →   1 ? genPos + 1
```

or also

```
instanceOfInt →  gen(0)
gen(x) → if x>=0 then x ? gen(-(x+1))
                 else x ? gen(-x)
```
$\square$

In the following, we consider only ordinary rewrite systems over algebraic datatypes. For such systems, Definition 3 immediately implies the following property of `instanceOf`.

Lemma 3 (Completeness of generators). *For every ground constructor term t of sort S, there exists a rewrite sequence of* `instanceOf`S *to t.*

Proof. By structural induction on t. $\square$

The *XE* transformation replaces any free variable v in a term with an operation that evaluates to any value that could instantiate the variable v during a computation.

Definition 4 (Extra variable elimination). Let V be a set of (sorted) variables. Then the *instantiation substitution* IO_V is defined as

$$IO_V = \{x \mapsto \texttt{instanceOf}s_x \mid x \in V \text{ has sort } s_x\}$$

For every term t we define

$$XE(t) = IO_{\mathcal{V}ar(t)}(t)$$ $\square$

The following lemma extends Lemma 3 to terms with variables.

Lemma 4. *For every variable x and constructor term u of the same sort, $XE(x) \overset{*}{\to} XE(u)$.*

Proof. By structural induction on u. $\square$

Definition 5 (Transformation from OIS into OIS$^-$). Let $\mathcal{R}$ be an *OIS* program. We define $XE(\mathcal{R}) = \mathcal{R}' \cup I$, where I defines a fresh symbol $\texttt{instanceOf}S$ for every sort S in the signature of $\mathcal{R}$, and $l \to r'$ is a rule of $\mathcal{R}'$ iff $l \to r$ is a rule of $\mathcal{R}$ and $r' = IO_{\mathcal{E}var(l \to r)}(r)$. $\square$

Proposition 3. *If $\mathcal{R}$ is an overlapping inductively sequential TRSs, then $XE(\mathcal{R})$ is an overlapping inductively sequential TRSs with no extra variables.*

Proof. Let $XE(\mathcal{R}) = \mathcal{R}' \cup I$ according to the definition of of XE. The left-hand sides of $\mathcal{R}$ and $\mathcal{R}'$ are the same and the left-hand sides of I are constants, hence $XE(\mathcal{R})$ is overlapping inductively sequential. Furthermore, there are no extra variables in $\mathcal{R}'$ by construction and in I by definition. Hence, the proposition holds. $\square$

To claim the correctness of the XE transformation, we need to show that, under appropriate conditions and qualifications, every computation in the original system has a corresponding computation in the transformed system and vice versa. First, we discuss the completeness of XE. We state the completeness for narrowing derivations that compute constructor substitutions.

Lemma 5 (Completeness of XE steps). *Let $\mathcal{R}$ an* OIS *program. For all terms t, u, if $t \rightsquigarrow_\sigma u$ w.r.t. $\mathcal{R}$ where σ is a constructor substitution, then $XE(t) \xrightarrow{+} XE(u)$ w.r.t. $XE(\mathcal{R})$.*

Proof. Let $\mathcal{R}' = XE(\mathcal{R})$. Since rewriting is closed under context, an immediate consequence of Lemma 4 ensures that $XE(t) \xrightarrow{*} XE(\sigma(t))$ w.r.t. $\mathcal{R}'$. By the definition of a narrowing step, there is a variant $l \to r$ of a rule in $\mathcal{R}$, a position p of t and a substitution η with $\mathcal{D}om(\eta) \subseteq \mathcal{V}ar(l)$ such that $\sigma(t)|_p = \eta(l)$ and $u = \sigma(t)[\eta(r)]_p$. By definition of XE, $l \to r'$ is a variant of a rule in $\mathcal{R}'$ with $r' = IO_{\mathcal{E}var(l \to r)}(r)$. Thus, $\sigma(t) \to \sigma(t)[\eta(r')]_p$ is a rewrite step w.r.t. $\mathcal{R}'$ and, since rewriting is closed under instantiation, $XE(\sigma(t)) \to XE(\sigma(t)[\eta(r')]_p)$ w.r.t. $\mathcal{R}'$. Since $r' = IO_{\mathcal{E}var(l \to r)}(r)$, the instantiated terms $\eta(r')$ and $\eta(r)$ differ only in the instantiation of extra variables: they are instantiated to `instanceOf` operations in $\eta(r')$ whereas in $\eta(r)$ they are unbound. By definition of XE, they are bound in $XE(\eta(r))$ to the same `instanceOf` operations as in $XE(\eta(r'))$. Thus, we obtain

$$XE(t) \xrightarrow{*} XE(\sigma(t)) \to XE(\sigma(t)[\eta(r')]_p) = XE(\sigma(t)[\eta(r)]_p) = XE(u) \qquad \square$$

Lemma 6 (Completeness of XE derivations). *Let $\mathcal{R}$ an* OIS *program. For any terms t and constructor term u, if $t \overset{*}{\rightsquigarrow} u$ w.r.t. $\mathcal{R}$ where the substitution of each narrowing step is a constructor substitution, then for any ground constructor instance v of u, $XE(t) \xrightarrow{*} v$ w.r.t. $XE(\mathcal{R})$.*

Proof. If $t \overset{*}{\rightsquigarrow} u$ w.r.t. $\mathcal{R}$, Lemma 5 implies that $XE(t) \xrightarrow{*} XE(u)$ w.r.t. $XE(\mathcal{R})$. Let v be a ground constructor instance of u, i.e., there is a ground constructor substitution η with $\mathcal{D}om(\eta) = \mathcal{V}ar(u)$ and $\eta(u) = v$. Let x be a variable in $\mathcal{V}ar(u)$. By Lemma 3, $XE(x) \xrightarrow{+} \eta(x)$. Since the rewrite relation is closed under context, $XE(u) \xrightarrow{+} \eta(u)$ which proves the claim. $\square$

For narrowing derivations with arbitrary substitutions, the proof of this lemma fails since `instanceOf` rewrites only to constructor terms. To extend the proof to obtain a more general result, we need to consider a variation of `instanceOf` defined as follows:

$$\begin{array}{lll} \texttt{instanceOf}S & \to & s_1\texttt{(instanceOf}t_{11}\texttt{,...,instanceOf}t_{1a_1}\texttt{)} \\ & & \texttt{? ...} \\ & & \texttt{? } s_n\texttt{(instanceOf}t_{n1}\texttt{,...,instanceOf}t_{na_n}\texttt{)} \end{array}$$

where $\{s1, \ldots, s_n\}$ are all the signature symbols of sort S and the arguments of s_i have sorts $t_{i1}, \ldots, t_{ia_i}$. However, this extension is not relevant in practice since narrowing strategies used in functional logic languages compute only constructor substitutions [6, 7].

In general, the transformation *XE* is not sound, i.e., there are rewrite derivations in the transformed system that have no correspondence in the original system.

Example 6. Consider the following program defining an operation that evaluates to an arbitrary even number:

```
even → x+x
```

Applying *XE* to this program yields:

```
even → instanceOfInt + instanceOfInt
```

Consequently, the term `even` can be evaluated as follows:

$$\texttt{even} \to \texttt{instanceOfInt + instanceOfInt} \overset{+}{\to} \texttt{0 + 1} \to \texttt{1} \qquad \square$$

This examples shows that all the occurrences of an `instanceOf` operation originating from the same variable should be reduced to the same value. Derivations where this condition is satisfied are called *admissible.* We will show that the *XE* transformation is sound for admissible derivations.

The problem in the previous example would be eliminated by having only one occurrence of `instanceOfInt`. Therefore, we introduce a notation of terms where only one occurrence is represented so that the derivation above is no longer possible. Our notation uses pairs $\langle t, \chi \rangle$ of a term t and a substitution χ which represents the term $\chi(t)$. The substitution χ will be defined as $IO_{\mathcal{V}ar(t)}$ so that it contains a single occurrence of an `instanceOf` operation for each free variable of t. We define rewrite steps on this representation. A redex may occur in either t or χ. Rewriting in t corresponds to standard rewriting, whereas a rewrite step in χ may correspond to a multistep [22] in $\chi(t)$ if the bound variable has several occurrences in t.

Definition 6 (Transformation to term/substitution pairs). For every term t we define $XEP(t) = \langle t, IO_{\mathcal{V}ar(t)} \rangle$. For every *OIS* program $\mathcal{R}$ we define $XEP(\mathcal{R}) = \mathcal{R}' \cup I$, where I is as in Definition 5, and $l \to r'$ is a rule of $\mathcal{R}'$ iff $l \to r$ is a rule of $\mathcal{R}$ and $r' = \langle r, IO_{\mathcal{E}var(l \to r)} \rangle$. $\square$

Next we define rewrite steps on the pair representation of terms.

Definition 7 (Rewriting on term/substitution pairs). Let $\mathcal{R}$ be an *OIS* program and $XEP(\mathcal{R}) = \mathcal{R}' \cup I$. Let t be a term and $XEP(t) = \langle t, \chi \rangle$. We define a rewrite step on $XEP(t)$ as follows. $\langle t, \chi \rangle \to \langle t', \chi' \rangle$ if one of the following conditions holds:

(type-1 step) there exist a position p in t, a variant $l \to \langle r, \psi \rangle$ with fresh variables of a rule in $\mathcal{R}'$, a substitution σ such that $\mathcal{D}om(\sigma) \subseteq \mathcal{V}ar(l)$, $\sigma(l) = t|_p$, $t' = t[\sigma(r)]_p$, and $\chi' = \chi|_{\mathcal{V}ar(t')} \cup \psi$

(type-2 step) there exist a variable $v \in \mathcal{D}om(\chi)$ with $\chi(v) = \texttt{instanceOf}S$ and a rule

$$\texttt{instanceOf}S \to c(\texttt{instanceOf}S_1, \ldots, \texttt{instanceOf}S_k)$$

according to Definition 3 such that $t' = \{v \mapsto c(v_1, \ldots, v_k)\}(t)$, $\chi' = (\chi \backslash \{v \mapsto \texttt{instanceOf}S\}) \cup \{v_i \mapsto \texttt{instanceOf}S_i \mid i = 1, \ldots, k\}$ where $v_1, \ldots, v_k$ are fresh variables. □

The following proposition states an important invariant of a rewrite sequence on term/substitution pairs.

Proposition 4. *Let $\mathcal{R}$ be an* OIS *program, $\mathcal{R}' = XEP(\mathcal{R})$, t a term in $\mathcal{R}$ and $t' = XEP(t)$. If $t' \to \langle s, \chi \rangle$ is a rewrite step in $\mathcal{R}'$, then $\mathcal{D}om(\chi) = \mathcal{V}ar(s)$ and $\chi(x) = \texttt{instanceOf}S$ for all variables $x \in \mathcal{V}ar(s)$ of sort S. In particular, there exists a term u in $\mathcal{R}$ such that $XEP(u) = \langle s, \chi \rangle$.*

Proof. Direct from Definition 7. □

Our notion of rewrite step on the term/substitution pair representation of a term t directly corresponds to the notion of multistep of t in standard rewriting. A multistep can be seen as an order-independent sequence of steps. In our particular case, every occurrence of $\texttt{instanceOf}S$ resulting from the same variable is replaced by the same replacement. Therefore, the term/substitution pair representation ensures that only admissible reduction sequences are computed. The following lemma formalizes this correspondence.

Lemma 7. *Let $\mathcal{R}$ be an* OIS *program, t a term and $XEP(t) = \langle t, \chi \rangle$. For every step $\langle t, \chi \rangle \to \langle t', \chi' \rangle$ w.r.t. $XEP(\mathcal{R})$ there exists a rewrite derivation $\chi(t) \xrightarrow{+} \chi'(t')$ w.r.t. $XE(\mathcal{R})$.*

Proof. The proof is by cases on the step type of $\langle t, \chi \rangle \to \langle t', \chi' \rangle$.

Case type-1: By definition of type-1 step, there exists a position p of t and a rule $l \to r$ in $\mathcal{R}$ such that in $XEP(\mathcal{R})$, $l \to \langle r, \psi \rangle$ is a rule and $\sigma(l) = t|_p$ in $XEP(\mathcal{R})$ and $t' = t[\sigma(r)]_p$. Also note that $l \to r'$ is a rule of $XE(\mathcal{R})$ where, by Def. 5, $r' = \psi(r)$. Thus, in $XE(\mathcal{R})$, $\sigma(l) = \chi(t)|_p = \chi(t|_p)$ and $\chi(t) \to \chi(t)[\sigma(r')]_p = \chi(t[\sigma(r')]_p)$. We show that $\chi(t[\sigma(r')]_p) = \chi'(t') = \chi'(t[\sigma(r)]_p)$. Let q be the position of a variable $x \in \mathcal{V}ar(t')$ and let S_x be the sort of x. If $x \in Var(t)$, then, by the definition of χ', $\chi'(x) = \chi(x)$. Otherwise, $x \in \mathcal{V}ar(r)$. By definition of rewrite step, $\mathcal{D}om(\sigma) = \mathcal{V}ar(l)$, hence $\sigma(x) = x$. By the definition of χ', $\chi'(x) = \psi(x) = \texttt{instanceOf}S_x$. By Def. 5, $r'|_q = \texttt{instanceOf}S_x$, thus, in $XE(\mathcal{R})$ we have $\chi(t) \xrightarrow{+} \chi'(t')$.

Case type-2: Let v be the variable witnessing the definition of type-2 step and let Q be the set of occurrence positions of v in t. By definition of type-2 step, $\chi(v) = \texttt{instanceOf}S$ and there exists a rule $\texttt{instanceOf}S \to c(\texttt{instanceOf}S_1, \ldots, \texttt{instanceOf}S_k)$. Consider the derivation $\chi(t) \xrightarrow{*} u$ that reduces the subterm $\texttt{instanceOf}S$ of $\chi(t)$ for every position in Q with

$c(\texttt{instanceOf}S_1, \ldots, \texttt{instanceOf}S_k)$. We show that $u = \chi'(t')$. Let q be a position. If q is disjoint from any position in Q, then by construction $\chi(t|_q) = u$ and, by Def. 7, $\chi(t|_q) = \chi'(t'|_q)$. If q is in Q, then, by construction $u|_q = c(\texttt{instanceOf}S_1, \ldots, \texttt{instanceOf}S_k)$ and by Def. 7, $t'|_q = c(v_1, \ldots, v_k)$ and $\chi'(v_i) = \texttt{instanceOf}S_i$ for all $i \in \{1, \ldots, k\}$ which implies $\chi'(t')|_q = c(\texttt{instanceOf}S_1, \ldots, \texttt{instanceOf}S_k)$. Thus, $u = \chi'(t')$ for every position, and in $\mathit{XE}(\mathcal{R})$ we have $\chi(t) \xrightarrow{+} \chi'(t')$. □

The term/substitution representation is an appealing formalism for this problem because it can be directly mapped to `let` binding constructs available in many programming languages. For instance, the transformed program of Example 6 can be coded in Curry [21] with a `let` binding as

```
even = let x = instanceOfInt in x+x
```

The semantics of the `let` binding construct is defined in such a way that all occurrences of `let` bound variables are replaced by the same replacement [1, 23] (efficiently implemented by sharing). Our notion of rewriting is a natural adaptation of this semantics.

In order to prove the soundness of the *XEP* transformation, we state some useful properties of rewrite steps on the term/substitution representation.

Lemma 8. *Let $\mathcal{R}$ be an OIS TRS, $\mathcal{R}' = \mathit{XEP}(\mathcal{R})$, t a term of $\mathcal{R}$, and $t' = \mathit{XEP}(t)$. If $t' \xrightarrow{+} u$ is a derivation with only type-2 steps, then there exists a constructor substitution σ such that $\mathit{XEP}(\sigma(t)) = u$.*

Proof. The claim for each step of the derivation is immediate from the definitions of type-2 step and *XEP*. □

Lemma 9. *Let $\mathcal{R}$ be an OIS TRS, $\mathcal{R}' = \mathit{XEP}(\mathcal{R})$, t a term of $\mathcal{R}$, and $t' = \mathit{XEP}(t)$. If $t' \to u'$ with a type-1 step, then there exists a term u such that $t \to u$ w.r.t. $\mathcal{R}$ and $\mathit{XEP}(u) = u'$.*

Proof. Let $t' \to u'$ be a type-1 rewrite step. By definition of type-1 steps, $t' = \langle t, \mathit{IO}_{\mathcal{V}ar(t)} \rangle$, there exist a position p in t, a variant $l \to \langle r, \psi \rangle$ with fresh variables of a rule in $\mathcal{R}'$, a substitution σ such that $\sigma(l) = t|_p$, $u = t[\sigma(r)]_p$, $\chi = \mathit{IO}_{\mathcal{V}ar(t)}|_{\mathcal{V}ar(u)} \cup \psi$, and $u' = \langle u, \chi \rangle$. Furthermore, $l \to r \in \mathcal{R}$ and $\psi = \mathit{IO}_{\mathcal{E}var(l \to r)}$. Hence, $t \to u$ is a rewrite step w.r.t. $\mathcal{R}$ and $\mathit{XEP}(u) = \langle u, \mathit{IO}_{\mathcal{V}ar(u)} \rangle$. Finally, $\chi = \mathit{IO}_{\mathcal{V}ar(t)}|_{\mathcal{V}ar(u)} \cup \psi = \mathit{IO}_{\mathcal{V}ar(t)}|_{\mathcal{V}ar(u)} \cup \mathit{IO}_{\mathcal{E}var(l \to r)} = \mathit{IO}_{\mathcal{V}ar(u)}$ (since $\mathcal{D}om(\sigma) \subseteq \mathcal{V}ar(l)$ by definition of type-1 step), which proves the claim. □

Now we are ready to state the soundness of the *XEP* transformation.

Lemma 10 (Soundness of *XEP*). *Let $\mathcal{R}$ be an OIS TRS, $\mathcal{R}' = \mathit{XEP}(\mathcal{R})$, and t a term of $\mathcal{R}$. If there exists a derivation $t' \xrightarrow{*} \langle v, \nu \rangle$ w.r.t. $\mathcal{R}'$, where $t' = \mathit{XEP}(t)$, then there exists a term u such that $t \overset{*}{\rightsquigarrow} u$ in $\mathcal{R}$ with $u \leq \nu(v)$.*

Proof. First we prove the following auxiliary statement: For every derivation

$$t_0 \to t_1 \to \cdots \to t_n$$

w.r.t. $\mathcal{R}'$ where $t_0 = \mathit{XEP}(t)$ and the last step is a type-1 step, there exists a term u with $t \overset{*}{\rightsquigarrow} u$ in $\mathcal{R}$ and $\mathit{XEP}(u) = t_n$. We prove this statement by induction on the number k of type-1 steps in this derivation.

Base case ($k = 1$): Since the final step is the only type-1 step, all steps $t_i \to t_{i+1}$ ($i \in \{0, \ldots, n-1\}$) are type-2 steps. Lemma 8 implies the existence of a constructor substitution σ with $\mathit{XEP}(\sigma(t)) = t_{n-1}$. Furthermore, Lemma 9 applied to the type-1 step $t_{n-1} \to t_n$ implies the existence of a term u such that $\sigma(t) \to u$ w.r.t. $\mathcal{R}$ and $\mathit{XEP}(u) = t_n$. Since σ is a constructor substitution, $t|_p$ is not a variable for the redex position p used in this rewrite step. Thus, by definition of narrowing, $t \rightsquigarrow u$.

Inductive case ($k > 1$): Let $t_0 \to t_1 \to \cdots \to t_m$ be the initial derivation steps ($m < n$) such that $t_i \to t_{i+1}$ ($i \in \{0, \ldots, m-1\}$) are type-2 steps and $t_{m-1} \to t_m$ is a type-1 step. Similarly to the base case, there exists a term u such that $t \rightsquigarrow u$ w.r.t. $\mathcal{R}$ and $\mathit{XEP}(u) = t_m$. Applying the induction hypothesis to the remaining derivation

$$t_m \to t_{m+1} \to \cdots \to t_n$$

implies the existence of a term u' such that $u \overset{*}{\rightsquigarrow} u'$ w.r.t. $\mathcal{R}$ and $\mathit{XEP}(u') = t_n$. Thus, $t \overset{*}{\rightsquigarrow} u'$ w.r.t. $\mathcal{R}$.

Now, consider a derivation

$$t_0 \to t_1 \to \cdots \to t_m \to \cdots \to t_n$$

w.r.t. $\mathcal{R}'$ such that $\mathit{XEP}(t) = t_0$, $t_{m-1} \to t_m$ is the last type-1 step of this derivation. Our auxiliary statement shows the existence of a term u with $t \rightsquigarrow u$ w.r.t. $\mathcal{R}$ and $\mathit{XEP}(u) = t_m$. Lemma 8 applied to the remaining type-2 steps $t_i \to t_{i+1}$ ($i \in \{m, \ldots, n-1\}$) implies the existence of a substitution σ with $\mathit{XEP}(\sigma(u)) = t_n = \langle v, \nu \rangle$. By definition of XEP, $\sigma(u) = v$. Hence, $u \leq v \leq \nu(v)$. □

The following lemma shows the completeness of single *XEP* steps.

Lemma 11. *Let $\mathcal{R}$ an* OIS *program and t a term, and σ a constructor substitution in $\mathcal{R}$. Then there exists a derivation $\mathit{XEP}(t) \overset{*}{\to} \mathit{XEP}(\sigma(t))$ w.r.t. $\mathit{XEP}(\mathcal{R})$.*

Proof. The lemma vacuously holds for $\sigma = \varnothing$ or $\sigma(t) = t$. First we prove the claim for a particular form of σ which instantiates a single variable with a single constructor symbol. Thus, consider $\sigma = \{x \mapsto c(x_1, \ldots, x_n)\}$, $n \geq 0$, with $x \in \mathcal{V}ar(t)$. Let $\mathit{XEP}(t) = \langle t, \chi \rangle$. Since $\chi(x) = \texttt{instanceOf}S$, where S is the sort of x, $\mathit{XEP}(t) \to \langle \sigma(t), \chi \backslash \sigma \cup \{x_i \mapsto \texttt{instanceOf}S_i \mid i = 1, \ldots, n\} \rangle$ by definition of type-2 step. Since $\mathcal{V}ar(\sigma(t)) = (\mathcal{V}ar(t) \backslash \{x\}) \cup \{x_1, \ldots, x_n\}$, the claim follows by definition of $\mathit{XEP}(\sigma(t))$.

For the general case of σ, the claim follows by induction on the structure of $\sigma(x)$ and by induction on the number of variables in $\mathcal{D}om(\sigma)$. □

Lemma 12 (Completeness of *XEP* steps). *Let $\mathcal{R}$ an* OIS *program. For all terms t, u, if $t \rightsquigarrow_\sigma u$ w.r.t. $\mathcal{R}$ where σ is a constructor substitution, then $\mathit{XEP}(t) \overset{+}{\to} \mathit{XEP}(u)$ w.r.t. $\mathit{XEP}(\mathcal{R})$.*

Proof. Let $\mathcal{R}' = \mathit{XEP}(\mathcal{R})$. Due to Lemma 11, $\mathit{XEP}(t) \xrightarrow{*} \mathit{XEP}(\sigma(t))$ w.r.t. $\mathcal{R}'$. By the definition of a narrowing step, there is a variant $l \to r$ of a rule in $\mathcal{R}$, a position p of t and a substitution η with $\mathcal{D}om(\eta) \subseteq \mathcal{V}ar(l)$ such that $\sigma(t)|_p = \eta(l)$ and $u = \sigma(t)[\eta(r)]_p$. By definition of *XEP*, $l \to \langle r, IO_{\mathcal{E}var(l \to r)}\rangle$ is a variant of a rule in $\mathcal{R}'$. Let $\mathit{XEP}(\sigma(t)) = \langle \sigma(t), \chi\rangle$. By definition of type-1 step, $\mathit{XEP}(\sigma(t)) \to \langle u, \chi|_{\mathcal{V}ar(u)} \cup IO_{\mathcal{E}var(l \to r)}\rangle$. Since $\mathcal{V}ar(u) = \mathcal{V}ar(\sigma(t))|_{\mathcal{V}ar(u)} \cup \mathcal{E}var(l \to r)$ and $\mathcal{D}om(\chi) = \mathcal{V}ar(\sigma(t)) \supseteq \mathcal{V}ar(\sigma(t))|_{\mathcal{V}ar(u)}$, we have $IO_{\mathcal{V}ar(u)} = \chi|_{\mathcal{V}ar(u)} \cup IO_{\mathcal{E}var(l \to r)}$. Hence, $\mathit{XEP}(\sigma(t)) \to \mathit{XEP}(u)$ which proves the claim. □

The following theorem summarizes the main results of this section—the correctness of the *XEP* transformation.

Theorem 2 (Correctness of *XEP*). *Let $\mathcal{R}$ be a OIS TRS, $\mathcal{R}' = \mathit{XEP}(\mathcal{R})$, t, s terms of $\mathcal{R}$, and $t' = \mathit{XEP}(t)$. Then the following claims hold.*

Soundness *If $t' \xrightarrow{*} \langle v, \nu\rangle$ is a derivation w.r.t. $\mathcal{R}'$, then there exists a narrowing derivation $t \overset{*}{\rightsquigarrow} u$ w.r.t. $\mathcal{R}$ with $u \leq \nu(v)$. In particular, if $\nu(v)$ is a constructor term, then $\nu = \varnothing$ and u is a constructor term.*

Completeness *If $t \overset{*}{\rightsquigarrow} s$ w.r.t. $\mathcal{R}$, then there exists a derivation $t' \xrightarrow{*} s'$ w.r.t. $\mathcal{R}'$ such that $s' = \mathit{XEP}(s)$. In particular, if s is a constructor term, then there exists a derivation $t' \xrightarrow{*} \langle v, \varnothing\rangle$ w.r.t. $\mathcal{R}'$ for any ground constructor instance v of s.*

Proof. The soundness follows from Lemma 10. If ν is not empty, $\nu(v)$ is not a constructor term by Proposition 4.

The completeness follows from Lemma 12 by induction on the length of the derivation. If s is a constructor term and η is a ground constructor substitution, Lemma 11 implies the existence of a derivation $\mathit{XEP}(s) \xrightarrow{*} \mathit{XEP}(\eta(s)) = \langle \eta(s), \varnothing\rangle$. □

The correctness of the *OE* transformation is proved using the needed narrowing strategy, whereas the correctness of *XEP* makes no assumption on the strategy. We use needed narrowing in the first case since we do not see an easy proof for a more general strategy. Since in practice one wants to use an efficient strategy, our choice is not limiting. Likewise, proving the correctness of *XEP* for needed narrowing does not seem easy. However, we remark the tight correspondence of steps in a TRS $\mathcal{R}$ and the transformed TRS $\mathit{XEP}(\mathcal{R})$ shown in Lemmas 9 and 12. The difference between the steps in the two systems is only in type-2 steps. These steps are avoided by the original TRS which performs narrowing steps as opposed to rewrite steps. Narrowing steps are more expensive due to the computation of variable instantiations. Since these instantiations correspond to type-2 steps, we conjecture that the costs of both derivations in a practical implementation are comparable.

The above results show that, loosely speaking, variables and overlapping rules have the same computational power in a functional logic language. The evaluation of expressions with free variables, particularly in the tradition of logic programming, produces variable bindings. These bindings are an integral part of the computation. This information seems to be lost with the *XEP* transformation. A simple way to recover this information is to transform the initial term t of a computation into a tuple $(t, x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are the variables of t. The evaluation of the tuple will be $(e, b_1, \ldots, b_n)$ where e is the computed value and $b_1, \ldots, b_n$ constitute the computed answer. A remaining obstacle is that bindings may contain variables whereas in our approach $b_1, \ldots, b_n$ are ground. To overcome this obstacle, one may adopt the convention that an occurrence of `instanceOf` is only evaluated if its value is necessary to perform a type-1 step. Observe that type-1 steps are never performed in $b_1, \ldots, b_n$.

5 Conclusion

We have presented two transformations on functional logic programs. The first transformation eliminates overlapping rules by introducing auxiliary functions and extra variables. Together with the results of [5], this transformation shows that any functional logic program can be mapped into an inductively sequential TRS with extra variables so that it can be executed by needed narrowing. Hence, the class ISX is a reasonable core language for functional logic programming. The second transformation completely eliminates logic variables from functional logic computations by replacing them with operations defined by overlapping rules. The correctness of this transformation requires the consistent evaluation of these new operations w.r.t. the logic variable occurrences. This can be achieved by sharing which is usually available in lazy languages.

The results presented in this paper provide a better understanding of the features of functional logic languages and their interactions. Although the source level of such languages extend purely functional languages by overlapping rules *and* extra variables, our results show that only one of these alternative concepts is enough for a core language.

Apart from these theoretical considerations, our results have also a practical interest since a simplified core language can reduce the implementation effort it requires. For instance, typical implementations of core languages are based on abstract machines that bridge the gap between the source level and the hardware (e.g., [8, 20, 25]). Usually, these machines provide instructions and data structures to support the implementation of both overlapping rules and logic variables. Our results enable the simplification of these abstract machines. For instance, specific instructions to handle computations that use overlapping rules need not be considered in an abstract machine if the *OE* transformation is applied in the compilation process. This is done in the implementations described in [14, 28], although without any formal justification. Likewise, the handling of logic variables (e.g., data structures such as binding arrays and binding instructions) can be removed if the *XEP* transformation is applied. Which of the two alternatives is more convenient depends on the concrete architecture of the machine. A simplified core language can also reduce the effort to build tools for functional logic languages. For instance, recent tools for debugging functional logic programs (e.g., tracers [13], profilers [12], slicers [27]) or program optimization (e.g., partial evaluation [2]) are based on a core language that supports both overlapping rules and logic variables which could be simplified using our results.

Finally, the *XEP* transformation also sheds some new light on the role of logic variables in declarative programming. It has been sometimes argued (in the functional programming community) that the instantiation of a logic variable during a computation is similar to a side effect due to its global visibility. For instance, this has led to the modeling of logic variables as references in Haskell [15]. However, our results show that the binding of a logic variable can be also interpreted as the stepwise evaluation of an operation so that the power of narrowing computations can be obtained by rewriting.

We have presented our results for a first-order many-sorted functional logic language. The extension to higher-order features is not difficult if one uses the well-known translation of higher-order functions into first-order rewrite rules by interpreting partial function applications as constructor terms and introducing a family of application operations for these terms (e.g., see [9, 29]). The extension to polymorphically typed languages is not so obvious since the *XEP* transformation assumes that the type of each logic variable is known at compile time. This information is always available in a many-sorted TRS but could be difficult to obtain in a polymorphic functional logic language where logic variables might have an arbitrary type. In this case, one could define a specific "polymorphic" `instanceOf` operation that evaluates to values of all possible types. However, this is not practical due to an increase of the search space size and the possibility of ill-typed expressions during a computation. An appropriate solution to this problem is a topic for future research.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, Vol. 40, No. 1, pp. 795–829, 2005.
2. E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluator for a Multi-Paradigm Declarative Language. *Journal of Functional and Logic Programming*, Vol. 2002, No. 1, 2002.
3. S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.
4. S. Antoy. Optimal Non-Deterministic Functional Logic Computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pp. 16–30. Springer LNCS 1298, 1997.
5. S. Antoy. Constructor-based Conditional Narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pp. 199–206. ACM Press, 2001.
6. S. Antoy. Evaluation Strategies for Functional Logic Programming. *Journal of Symbolic Computation*, Vol. 40, No. 1, pp. 875–903, 2005.
7. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
8. S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A Virtual Machine for Functional Logic Computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pp. 108–125. Springer LNCS 3474, 2005.
9. S. Antoy and A. Tolmach. Typed Higher-Order Narrowing without Higher-Order Strategies. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pp. 335–352. Springer LNCS 1722, 1999.
10. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
11. J.A. Bergstra and J.W. Klop. Conditional Rewrite Rules: Confluence and Termination. *Journal of Computer and System Sciences*, Vol. 32, No. 3, pp. 323–362, 1986.

12. B. Braßel, M. Hanus, F. Huch, J. Silva, and G. Vidal. Run-Time Profiling of Functional Logic Programs. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, pp. 182–197. Springer LNCS 3573, 2005.
13. B. Braßel, M. Hanus, F. Huch, and G. Vidal. A Semantics for Tracing Declarative Multi-Paradigm Programs. In *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'04)*, pp. 179–190. ACM Press, 2004.
14. B. Braßel and F. Huch. Translating Curry to Haskell. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pp. 60–65. ACM Press, 2005.
15. K. Claessen and P. Ljunglöf. Typed Logical Variables in Haskell. In *Proc. ACM SIGPLAN Haskell Workshop*, Montreal, 2000.
16. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pp. 243–320. Elsevier, 1990.
17. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A Logic plus Functional Language. *Journal of Computer and System Sciences*, Vol. 42, No. 2, pp. 139–185, 1991.
18. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
19. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
20. M. Hanus and R. Sadre. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming*, Vol. 1999, No. 6, 1999.
21. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
22. G. Huet and J.-J. Lévy. Computations in Orthogonal Rewriting Systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pp. 395–443. MIT Press, 1991.
23. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pp. 144–154. ACM Press, 1993.
24. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
25. W. Lux and H. Kuchen. An Efficient Abstract Machine for Curry. In K. Beiersdörfer, G. Engels, and W. Schäfer, editors, *Informatik '99 — Annual meeting of the German Computer Science Society (GI)*, pp. 390–399. Springer, 1999.
26. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, Vol. 12, pp. 191–223, 1992.
27. C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proc. of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*, pp. 123–134. ACM Press, 2004.
28. A. Tolmach, S. Antoy, and M. Nita. Implementing Functional Logic Languages Using Multiple Threads and Stores. In *Proc. of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pp. 90–102. ACM Press, 2004.
29. D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pp. 441–454, 1982.