On the Correctness of the Factoring Transformation

Pascual Julián Iranzo[1]

Departamento de Informática
Universidad de Castilla–La Mancha
Ciudad Real, Spain
pjulian@inf-cr.uclm.es

Abstract. Non-deterministic computations greatly enhance the expressive power of functional logic programs, but are often computationally expensive. Recently we analyzed a program transformation, called *factoring* transformation [4], that may save both execution time and memory occupation in a non-deterministic computation. In this paper we study the formal properties of our transformation, proving its correctness for constructor-based left linear rewriting systems, under some well established conditions. We also introduce some proof techniques that help us to reason with this kind of rewriting systems.

1 Introduction

The aim of functional logic programming is to integrate the best features of both functional and logic programming. Logic programming provides the use of predicates and logical formulas. Logic programs have a great expressive power thanks to the ability of logic languages of computing using logical variables, partial data structures and an operational mechanism that permits a non-deterministic search for answers. Functional programming is based on the concept of function. In a functional program functions are defined by means of equations. The deterministic evaluation of ground expressions increases the efficiency of functional programs. The concept of evaluation strategies, that also increase the efficiency of functional computations, relays on the existence and manipulation of nested terms. The combination of these features makes functional logic languages both more expressive than functional languages and more efficient than traditional logic languages.

Non-determinism is an essential feature of these integrated languages, since it allows problem solving using programs that are textually shorter, easier to

[1] Supported by the Spanish Research Funding Agency (CICYT) TIC 2001-2705-C03-01, by Acción Integrada Hispano-Italiana HI2000-0161, by Acción Integrada Hispano-Alemana HA2001-0059, and the Valencian Research Council under grant GV01-424.

understand and maintain, and more declarative than their deterministic counterparts (see [2], [9] or [14] for several motivating examples).

In a functional logic programming language, non-deterministic computations are modeled by the defined operations of a constructor-based left linear rewrite system [2, 8]. The following emblematic example [8, Ex. 2] defines an operation, *coin*, that non-deterministically returns either zero or one.

$$coin \to 0$$
$$coin \to s(0)$$

Rewrite systems with operations such as `coin` are non-confluent. A computation in these rewrite systems may have distinct normal forms and/or does not terminate. Non-determinism abstracts the choice of one of several outcomes of a computation. The outcome of a non-deterministic computation is selected somewhat randomly. For example, for the operation *coin* defined above, each solution, 0 or $s(0)$, is equally likely to be produced. There is no feasible means of deciding which replacement should be chosen at the time `coin` is evaluated. Therefore, evaluation under both replacements must be considered. In general, to ensure operational completeness, all the possible replacements of a non-deterministic computation must be executed fairly. In fact, if one replacement is executed only after the computation of another replacement is completed, the second replacement will never be executed if the computation originated by a first replacement does not terminate.

This approach, which we refer to as *fair independent computations*, captures the intended semantics, but clearly it is computationally costly. For example, consider the following operations, where "bigger" is a variant of `coin`:

$$add(0, Y) \to Y$$
$$add(s(X), Y) \to s(add(X, Y))$$
$$positive(0) \to false$$
$$positive(s(X)) \to true$$
$$bigger \to s(0)$$
$$bigger \to s(s(0))$$

The evaluation of the term $positive(add(bigger, 0))$ requires the evaluation of the subterm *bigger*. Therefore, one must compute fairly and independently both $positive(add(s(0), 0))$ and $positive(add(s(s(0), 0))$.

However, in a situation like the one just described, the cost of fair independent computations might be avoided by means of a programming technique that facilitates the use of deterministic choices. Our programming technique is based on the introduction of a new symbol, denoted by the infix operator " ! " and read as *alternative*, into the signature of the TRS modeling a functional logic program. We treat the new symbol as a polymorphic operation[1], defined by the rules [8, 14]:

$$ALT1 : X!Y \to X$$
$$ALT2 : X!Y \to Y$$

[1] The reader can see another different approach in [4], where we also consider the alternative symbol as an overloaded constructor.

The *alternative* operation allows us to give an equivalent definition of the operation `bigger`:

$$bigger \to s(0!s0)$$

where a common context of the right-hand sides of the rules defining the operation *bigger*, in the original term rewriting system, has been "factored". The advantage of this new definition of *bigger* with respect to the original one is that, using a *needed* strategy [2] or the constructor-based lazy narrowing of [9], only the factored portion of the two alternative right-hand sides of the rewrite rules of *bigger* is needed by a context and no fair independent computations are created. As it is shown by the following (needed) derivation:

$$\begin{aligned} positive(add(bigger, 0)) &\to positive(add(s(0!s(0)), 0) \\ &\to positive(s(add(0!s(0)), 0)) \\ &\to true. \end{aligned}$$

where two non-deterministic fair independent computations have been merged. Roughly speaking, if we "factor out" the common part of a set of non-deterministic choices, when it is possible, we can avoid the replication of the choice's context obtaining a gain. For this example, the cost criteria developed in [4] reveal that our programming technique cuts the number of steps in half and reduces the memory consumption by 25% w.r.t. a computation performed using the original program. On the other hand, in cases where factoring the right-hand sides of two rewrite rules does not eliminate the need of fair independent computations, the run-time cost of factoring is a single additional rewrite step. For realistic programs, this cost is negligible. Hence, the factorization of right-hand sides is a source of potential improvement. In the best case, it saves computing time and/or storage for representing terms. In the worst case, it costs one extra step and very little additional memory (see [4] for some examples on larger benchmark programs, coded using the functional logic language Curry [11], that illustrate in more detail the impact on the efficiency produced by our factoring technique).

We can see all this process as a program transformation that starting from an original program produce a new and, in many cases, more efficient program. In this paper, we define our program transformation in a formal setting and we prove its correctness under precise conditions. That is, we prove that the original and the transformed program are semantically equivalent. In order to obtain this result it is convenient to introduce some proof techniques that help us to reason with non-confluent constructor-based rewrite systems, where the concept of *descendants* of a redex [12] is not well established.

The plan of the paper is as follows: Section 2 introduces some preparatory concepts that are used in the rest of the paper. Section 3 gives a formal definition of our transformation. Section 4 discusses the conditions under our transformation is sound and complete and we introduce some concepts to facilitate our proofs: namely, an embedding relation, that abstracts the intuitive notion of "containment" between terms with alternative symbols, and the concept of non-factorized program, that defines a standard form of program. Finally, Section 5 contains our conclusions.

A full version with missing proofs can be found as a wed document at the URL address `http://www.inf-cr.uclm.es/www/pjulian`.

2 Preliminaries

In this section we briefly summarize some well-known notations and concepts about term rewriting systems [5] and functional logic programming [10].

2.1 Terms, substitutions and positions.

Throughout this paper, $\mathcal{X}$ denotes a countably infinite set of *variables* and $\mathcal{F}$ denotes a set of *function symbols* (also called the *signature*), each of which has a fixed associated arity. We assume that the signature $\mathcal{F}$ is partitioned into two disjoint sets $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions.* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denotes the set of *terms* or *expressions* built from $\mathcal{F}$ and $\mathcal{X}$. $\mathcal{T}(\mathcal{F})$ denotes the set of *ground terms*, while $\mathcal{T}(\mathcal{C}, \mathcal{X})$ denotes the set of *constructor* terms. If $t \notin \mathcal{X}$, then $\mathcal{R}oot(t)$ is the function symbol heading the term t, also called the *root symbol* of t. A term t is *linear* if t does not contain multiple occurrences of the same variable. $\mathcal{V}ar(o)$ is the set of variables occurring in the syntactic object o.

A *substitution* is a mapping from the set of variables $\mathcal{X}$ to the set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that its *domain* $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is finite. We frequently identify a substitution σ with the set $\{x/\sigma(x) \mid x \in \mathcal{D}om(\sigma)\}$. We denote the identity substitution by id. We say that σ is a *constructor substitution* if $\sigma(x)$ is constructor term for each $x \in \mathcal{D}om(\sigma)$. We define the composition of two substitutions σ and θ, denoted $\sigma \circ \theta$ as usual: $\sigma \circ \theta(x) = \hat{\sigma}(\theta(x))$, where $\hat{\sigma}$ is the extension of σ to the domain of the terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We say that θ is *more general* than σ (in symbols $\theta \leq \sigma$) iff $(\exists\gamma)\ \sigma = \gamma \circ \theta$. The *restriction* $\sigma_{\restriction\mathcal{V}}$ of a substitution σ to a set $\mathcal{V}$ of variables is defined by $\sigma_{\restriction\mathcal{V}}(x) = \sigma(x)$ if $x \in \mathcal{V}$ and $\sigma_{\restriction\mathcal{V}}(x) = x$ if $x \notin \mathcal{V}$. We write $\sigma = \theta[\mathcal{V}]$ iff $\sigma_{\restriction\mathcal{V}} = \theta_{\restriction\mathcal{V}}$. A *renaming* is a substitution ρ such that there exists the inverse substitution ρ^{-1} and $\rho \circ \rho^{-1} = \rho^{-1} \circ \rho = id$. Given a set S of terms and a substitution σ, $\sigma(S) = \{\sigma(t) \mid t \in S\}$.

A term t is *more general* than s (or s is an *instance* of t), in symbols $t \leq s$, if $(\exists\sigma)\ s = \sigma(t)$. Two terms t and t' are *variants* if there exists a renaming ρ such that $t' = \rho(t)$. A *unifier* of a pair of terms $\langle t_1, t_2 \rangle$ is a substitution σ such that $\sigma(t_1) = \sigma(t_2)$. A unifier σ is called *most general unifier* (mgu) if $\sigma \leq \sigma'$ for every other unifier σ'.

Positions of a term t (also called *occurrences*) are represented by sequences of natural numbers used to address subterms of t. The concatenation of the sequences p and w is denoted by $p.w$. We let Λ denote the empty sequence. $\mathcal{P}os(t)$ and $\mathcal{FP}os(t)$ denote, respectively, the set of positions and the set of nonvariable positions of the term t. If $p \in \mathcal{P}os(t)$, $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the result of replacing the subterm $t|_p$ by the term s.

2.2 Term rewriting, classes of term rewrite systems and programs.

We limit the discussion to unconditional term rewriting systems. A *rewrite rule* is a pair $l \to r$ with $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. l and r are called the *left-hand side* (lhs) and *right-hand side* (rhs) of the rewrite rule, respectively. A *term rewriting system* (TRS) $\mathcal{R}$ is a finite set of rewrite rules.

Rewrite rules in a TRS define a *rewriting relation* $\to$ between terms which can be defined as follows: $t \to_{p,l\to r} s$ if there exists a position $p \in \mathcal{P}os(t)$, a rewrite rule $l \to r$, and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. We say that s is reduced to t in a *rewrite step*, $t \to_{p,l\to r} s$ and the subterm $t|_p = \sigma(l)$ is a *redex* (*red*ucible *ex*pression) of t. A sequence of (zero or more) rewriting steps is denoted by $(t \to^* s)$ $t \to^+ s$. A term t is in *normal form* if t is a term without redexes. A term s has a normal form if there exists a reduction sequence $s \to^* t$, where t is a normal form.

A TRS is *terminating* or *noetherian* if there are no infinite reduction sequences. Since in this work we do not impose the requirement of terminating rules, normal forms may not exist.

A TRS is called *confluent* if, whenever a term s reduces to two terms t_1 and t_2, both t_1 and t_2 reduce to the same term. Confluence is a decidable property for terminating TRSs. Confluent TRSs have unique normal forms, when they exist. The confluence property is lost when non convergent critical pairs appear. Given two rules $l_1 \to r_1$ and $l_2 \to r_2$ if there exists a position $p \in \mathcal{FP}os(l_1)$ such that $l_1|_p$ and l_2 unify with mgu σ, then the pair of terms $\langle \sigma(r_1), \sigma(l_1)[\sigma(r_2)]_p \rangle$ is a *critical pair*. If two rules have a critical pair, we say that they *overlap*. A special case of overlapping rules are, those what we call *variant overlapping* rules, that is, overlapping rules whose lhss are variants one of each other [2]. We call a TRS with overlapping rules *overlapping* TRS.

A TRS is said to be *constructor-based* (CB) if the "arguments" of the lhs of its rules are constructor terms or variables.

A TRS is said to be *left-linear* (resp. *rigth-linear*) if for each rule $l \to r$ in the TRS, the lhs l (resp. rhs r) is a linear term.

Since, in this paper, we are interested in non-determinism we are going to work with overlapping CB and left linear TRSs that we assimilate to *programs*. This is a suitable class of programs for integrating functional and logic languages and modeling a non-deterministic behavior [2, 8].

2.3 Functional Logic Programming.

Functional logic languages can be considered as an extension of functional languages with principles derived from logic programming. Most of this languages use TRSs as programs and (some variant) of narrowing as operational semantics. Narrowing generalizes the *rewriting* operational mechanism of functional

[2] Rules of this kind are not unusual in practical programs. This is the only kind of overlapping allowed in *overlapping inductively sequential* rewrite systems, a class that supports both non-deterministic computations and optimal lazy evaluation [2].

languages. The narrowing relation induced by a TRS is defined as follows: a term t is reduced to the term s in a *narrowing step*, denoted $t \leadsto_{[p,R,\sigma]} s$, if there exists a position $p \in \mathcal{FP}os(t)$, a variant program rule $R = (l \to r)$ and a unifier σ of the terms $t|_p$ and l, such that $s = \sigma(t[r]_p)$. We say that there exists a *narrowing derivation* from a term t to a term s, if there exists a sequence of narrowing steps $t = t_0 \leadsto_{[p_1,R_1,\sigma_1]} t_1 \leadsto_{[p_2,R_2,\sigma_2]} \cdots \leadsto_{[p_n,R_n,\sigma_n]} t_n = s$ and we write $t \leadsto^*_\sigma s$, where $\sigma = \sigma_n \circ \ldots \circ \sigma_2 \circ \sigma_1$. We say that the pair $\langle s, \sigma \rangle$ is the *outcome* of the derivation. Usually we say that the term s is the *result* of the derivation and the substitution σ is the *computed answer*. Mostly, we are interested in derivations to constructor terms, that we call *values*.

Intuitively, narrowing computes a suitable substitution, σ, which when it is applied to a term t, the term $\sigma(t)$ can be reduced on a rewriting step $\sigma(t) \to_{p,R} s$. Note that, usually, the substitution σ computed by a narrowing step is a most general unifier, i.e., $\sigma = mgu(\{l = t|_p\})$. However we have relaxed this restriction to support narrowing strategies, such as needed narrowing [3] or INS [2], that may compute substitutions which are not most general unifiers.

Narrowing provides completeness in the sense of logic programming (computation of answers) and also in the sense of functional programming (computation of values or normal forms).

3 The Factoring Transformation

The aim of *program transformation* [6, 18] is to derive a program semantically equivalent to the original program. More accurately, given an initial program $\mathcal{R}$, we want to derive a new program $\mathcal{R}'$ which computes the same results and answers as $\mathcal{R}$ for any input term, but with a better behavior w.r.t. some determined properties (usually, we want the transformed program may be executed more efficiently than the initial one).

We are interested in the transformation of overlapping, CB and left linear TRSs modeling functional logic programs with a non-deterministic behavior [2, 9]. Beginning from this point, by abuse of language, and in order to lighten our discourse, we use the word "program" as a synonym of "overlapping, CB and left linear TRS". In a program, variant overlapping rules can be "merged" and expressed in a more concise way by means of the introduction of the alternative operation " ! ". We consider the alternative operation " ! " as defined by the pair of variant overlapping rules introduced in Section 1. Also, we consider that this pair of rules is present in all our programs. This assumption is harmless, since the operation can be added to any program that does not already define it without changing the meaning of existing operations.

We formalize our transformation, intuitively introduced in Section 1, by means of the foollowing definitions.

Definition 1. *[Term factoring] Let t, u and v be terms. We say that t is a* product *of u and v if and only if one of the following conditions hold:*

1. $t = u \,!\, v$ *or* $t = v \,!\, u$.

2. $t = f(t_1, \ldots, t_n)$, *where* f *is a symbol of arity* n *and* $t_1, \ldots, t_n$ *are terms,* $u = f(u_1, \ldots, u_n)$, $v = f(v_1, \ldots, v_n)$, *where* $u_1, \ldots, u_n, v_1, \ldots, v_n$ *are terms, and for all* $i \in \{1, \ldots, n\} \backslash \{j\}$, *with* $j \in \{1, \ldots, n\}$, $t_i = u_i = v_i$ *and* t_j *is a product of* u_j *and* v_j.

Conversely we say that u *and* v *are* factors *of* t.

Observe that the last case of Definition 1 may apply when f is " ! ".

Example 1. Both $s(0) \,!\, s(s(0))$ and $s(0 \,!\, s(0))$ are products of the terms $s(0)$ and $s(s(0))$. The term $f(X, s(0 \,!\, s(0)))$ is a product of the terms $f(X, s(0))$ and $f(X, s(s(0)))$. Note also that, for the terms $f(0,0)$ and $f(s(0), s(0))$ it is impossible to construct a product other that the trivial $f(0,0) \,!\, f(s(0), s(0))$.

Intuitively, a product t is built by sharing a common context of u and v over a `single` common position of both u and v. In the simplest case the context is a vacuum context and the single common position is the top position of the factors.

Definition 2. *[Program factoring] Let* $\mathcal{R}$ *be a program defining the alternative operation " ! ", and* $l_1 \to r_1$ *and* $l_2 \to r_2$ *are variant overlapping rules of* $\mathcal{R}$. *Without loss of generality we assume that* $l_1 = l_2$ *(since renaming the variables of a rule does not change the rewrite relation). A program* $\mathcal{R}'$ factors $\mathcal{R}$ *if and only if either* $\mathcal{R}' = \mathcal{R} \backslash \{l_1 \to r_1, l_2 \to r_2\} \cup \{l_1 \to r\}$, *where* r *is a product of* r_1 *and* r_2, *or* $\mathcal{R}'$ *factors some program* $\mathcal{R}''$ *and* $\mathcal{R}''$ *factors* $\mathcal{R}$.

Informally, the factoring transformation can be seen as a process that starting with an overlapping, left linear and CB TRS (the original program) produce a new TRS (the transformed program) by application of a sequence of the following transformation steps:

1. `merging step`: we arbitrarily select two variant overlapping rules (different from *ALT*) $l \to u$ and $l \to v$ that are merged into a new single rule $l \to u \,!\, v$ and the old rules are erased;
2. `factoring step`: if possible, we push the root alternative operator down the term by factoring some common part of the rhs.

This transformation process always terminates, provided that we work with finite programs and terms. Although we let unspecified both the starting program and the final program, it should be intuitively clear that the deeper the alternative operator can be pushed down the right-hand side, the more likely it will be to replace two fair independent computations by a single computation.

Example 2. Considering once more again the program $\mathcal{R} = \{bigger \to s(0), bigger \to s(s(0))\}$ (augmented with the *ALT* rules), the free application of the aforementioned transformation steps leads to the following residual programs:

$$\mathcal{R}'_1 = \{bigger \to s(0) \,!\, s(s(0))\} \qquad \text{and} \qquad \mathcal{R}'_2 = \{bigger \to s(0 \,!\, s(0))\}.$$

Both $\mathcal{R}'_1$ and $\mathcal{R}'_2$ factors $\mathcal{R}$ (according to Definition 2), but only $\mathcal{R}'_2$ can effectively produce a gain by merging two fair independent computations into a single computation. Finally, it is worthy to say that $\mathcal{R}'_2$ factors $\mathcal{R}'_1$ and $\mathcal{R}'_2$ cannot be further transformed.

Note that, since we introduce the *ALT* rules in the transformed program as well as in the original one, both programs derive the same signature and can be used to reduce the same kind of terms. Therefore, it is not necessary any kind of renaming transformation as it is the case of other more complex transformation techniques, e.g., the partial evaluation transformation of [1].

4 Correctness of the Factoring Transformation

From an operational semantics point of view, we say that a transformation is *sound* when, given a term, the results and answers computed by the transformed program are exactly the same as the ones computed by the original program for that term. The definition of *completeness* is the reverse of the last concept.

Definition 3. *Let $\Im$ be a mapping from programs to programs. Let $\mathcal{R}$ and $\mathcal{R}'$ be programs (with the same signature) such that $\mathcal{R}' = \Im(\mathcal{R})$. $\Im$ is* complete *if and only if for every term t and value v, $t \leadsto^*_\sigma v$ in $\mathcal{R}$ implies $t \leadsto^*_{\sigma'} v$ in $\mathcal{R}'$ where $\sigma = \sigma'[\mathcal{V}ar(t)]$. $\Im$ is* sound *if and only if for every term t and value v, $t \leadsto^*_{\sigma'} v$ in $\mathcal{R}'$ implies $t \leadsto^*_\sigma v$ in $\mathcal{R}$ where $\sigma = \sigma'[\mathcal{V}ar(t)]$.*

In this section we are interested in the study of the correctness properties of the factoring transformation given by Definition 2.

4.1 Embedding relation

For the class of programs we are working on there is not a notion of descendants [12] of a redex. Therefore, it is difficult to reason about the correctness properties of our transformation. To facilitate our proofs of correctness we introduce an intuitive notion of ordering, for terms with alternative symbols, in which a term that is "contained" inside another is smaller than the other.

Definition 4 (embedding relation).
The embedding relation *$\trianglelefteq$ on terms in $\mathcal{T}(\mathcal{F},\mathcal{X})$ is defined as the smallest relation satisfying: $x \trianglelefteq x$ for all $x \in \mathcal{X}$, and $s \trianglelefteq t$, if and only if:*

1. *if $\mathcal{R}oot(s) \neq\ !$ and $t = u\ !\ v$ then $s \trianglelefteq u$ or $s \trianglelefteq v$;*
2. *if $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ then $s_i \trianglelefteq t_i$ for all $i = 1, \ldots, n$.*

The strict embedding *relation $\triangleleft$ is defined, in terms of the embedding relation $\trianglelefteq$, as follows: $s \triangleleft t$, if and only if $s \trianglelefteq t$ and $s \neq t$.*

The above definition differs from the homeomorphic embedding relation of [7] or [16]. Note also that, for the second case, the operation symbol f may be an alternative symbol.

Example 3. These terms are in the embedding relation: $0 \trianglelefteq 0\ !\ s(X)$; $f(a) \trianglelefteq a\ !\ f(a\ !\ b)$ and $0\ !\ f(X) \trianglelefteq (0\ !\ s(0))\ !\ f(X)$.

The embedding relation $\trianglelefteq$ is a partial order over the set $\mathcal{T}(\mathcal{F},\mathcal{X})$. Clearly, the relation is reflexive, transitive, and antisymmetric.

Definition 5 (minimal element).
Let t be a term, we call minimal element of t, a term u such that $u \trianglelefteq t$ and there no exists a term $v \trianglelefteq t$ such that $v \triangleleft u$

The notion of minimal element can be extended to substitutions in a natural way: a substitution σ is *minimal* w.r.t. the substitution θ iff $\mathcal{D}om(\sigma) = \mathcal{D}om(\theta)$ and, for all $x \in \mathcal{D}om(\sigma)$, $\sigma(x)$ is a minimal element of $\theta(x)$.

Lemma 1. *If s and t are terms such that $s \trianglelefteq t$, then there exists a reduction sequence from t to s using only ALT rules.*

Proof. By structural induction on t. We distinguish the following cases:

1. $t \in \mathcal{X}$ or t is a nullary symbol
   Then, by Definition 4, $s = t$ and the claim follows by vacuity.
2. $t = u \,!\, v$
   Assume $\mathcal{R}oot(s) \neq\ !$, otherwise we are in the third case of this proof. Now, by Definition 4, $s \trianglelefteq u$ or $s \trianglelefteq v$. Assuming that $s \trianglelefteq u$, by induction hypothesis $u \to_{ALT} \ldots \to_{ALT} s$ and we can built the reduction sequence:
   $$t = u \,!\, v \to_{ALT_1} u \to_{ALT} \ldots \to_{ALT} s$$
3. $t = f(t_1, \ldots t_n)$
   Then s must be rooted by the symbol f, otherwise s would not be embedded in t. Hence, $s = f(s_1, \ldots s_n)$ and $s_i \trianglelefteq t_i$ for all $i \in \{1, \ldots, n\}$. By the inductive hypothesis $t_i \to_{ALT} \ldots \to_{ALT} s_i$ for all $i \in \{1, \ldots, n\}$ and we can built the reduction sequence:
   $$\begin{aligned} t = f(t_1, \ldots t_n) &\to_{ALT} \ldots \to_{ALT} f(s_1, \ldots t_n) \to_{ALT} \ldots \\ &\to_{ALT} \ldots \to_{ALT} f(s_1, \ldots s_n) = s. \end{aligned}$$

Corollary 1. *If u is a minimal element of t, then u is a normal form of t w.r.t. reduction sequences using only ALT rules.*

Proof. Immediate by Definition 5 and Lemma 1.

4.2 Completeness

The following lemma establishes the precise relation between a step in the original program and a reduction sequence in the transformed one.

Lemma 2. *Let $\mathcal{R}$ be a program, $\mathcal{R}'$ be a program that factors $\mathcal{R}$. If there exists a reduction step $A = t \to s$ in $\mathcal{R}$ then there exists a reduction sequence $D' = t \to^+ s$ in $\mathcal{R}'$.*

Proof. Assume that the step A is performed with the rule $R = l \to r$ over the position p and $t|_p = \theta(l)$, i.e., $A = t \to_{p,R} t[\theta(r)]_p$. We consider the following cases:

1. R is a rule that also belongs to $\mathcal{R}'$. Then the claim immediately holds.

2. R was merged in a transformed rule $R' = l \to r'$ of $\mathcal{R}'$, where r is a factor of r'. Then $r \trianglelefteq r$ and by Lemma 1 there exists the reduction sequence $r' \to_{ALT} \ \ldots \ \to_{ALT} r$. Therefore we can construct the following reduction sequence in $\mathcal{R}'$

$$t \to_{p,R'} t[\theta(r')]_p \to_{ALT} \ \ldots \ \to_{ALT} t[\theta(r)]_p$$

Now it is immediate to establish the equivalence between reduction sequences leading to a value in both the original and the transformed program.

Proposition 1 (pre-completeness). *Let $\mathcal{R}$ be a program and $\mathcal{R}'$ be a program that factors $\mathcal{R}$. If there exists a reduction sequence $D = t \to^* s$ in $\mathcal{R}$, where s is a value then there exists a reduction sequence $D' = t \to^* s$ in $\mathcal{R}'$.*

Proof. By induction on the number n of steps of the reduction sequence D and Lemma 2.

Finally, the completeness of the transformation is an easy consequence of the correctness of narrowing.

Theorem 1. *The factoring transformation is complete.*

Proof. Let $\mathcal{R}$ be a program and $\mathcal{R}'$ be a program that factors $\mathcal{R}$. If there exists a derivation $t \leadsto^*_\sigma s$ in $\mathcal{R}$

$\Rightarrow \sigma(t) \to^* s$ in $\mathcal{R}$ (by soundness of narrowing)
$\Rightarrow \sigma(t) \to^* s$ in $\mathcal{R}'$ (by Proposition 1)
$\Rightarrow t \leadsto^*_{\sigma'} s$ in $\mathcal{R}'$ with $\sigma' = \sigma[\mathcal{V}ar(t)]$ (by completeness of narrowing).

Note that it was not necessary to impose additional requirements to the programs (such as right linearity) or to use a special semantics (see Section 4.4) in order to obtain the completeness result.

4.3 Soundness

As we are going to see, our transformation does not preserve the soundness property in all cases. The following example points out this drawback and permit us to understand the kind of restrictions that are necessary to introduce to preserve this property.

Example 4. Given the initial program $\mathcal{R}$

$R_1 : double(X) \to X + X$
$R_2 : f \to double(0) \qquad R_3 : f \to double(s(0))$

extended with the usual rules for the addition, and the input term f, only the following results are possible: $\{0, s(s(0))\}$. On the other hand, the factored version of $\mathcal{R}$, namely $\mathcal{R}'$, is

$R_1 : \ double(X) \to X + X$
$R_{23} : f \to double(0 \,!\, s(0))$

and it can compute the additional result s(0) (twice) for the input term f.

This example reveals that, in general, our transformation is unsound, because there exists derivations in $\mathcal{R}'$ that cannot be reproduced in $\mathcal{R}$. At the first sight, a possible solution for this drawback might be to restrict the kind of operations that can be used in the transformation. Note that, the operation *double*, defined by rule R_1, is not right linear and this is the main source of our problem: the subterm at position 1 of $double(0 \mathbin{!} s(0))$ is duplicate, introducing new alternatives that are not produced when the reduction of f is done using the original program $\mathcal{R}$. Therefore, the solution might be to impose as a restriction that only operations defined by right linear rules should be part of a context containing a product. Unfortunately, a stronger restriction, namely the right linearity of the program, must be imposed. Suppose that we change the rule R_1, in $\mathcal{R}$, by this right linear rule

$$R_1 : double(X) \to s(s(0)) \times X,$$

we also introduce in our program the suitable rules for the operation $\times$ and we transform the resulting program to obtain a new factored version of it. Then when we proceed with the computation of f, due to the lack of right linearity of one of the rules defining the operation $\times$, the same problem is reproduced again (the original program computes the results $\{0, s(s(0))\}$ whereas the transformed program computes the results $\{0, s(0), s(s(0))\}$).

For the class of programs we use, the concept of right linearity must be revised, considering that the variables occurring in different alternatives of a rhs of a variant overlapping rule are completely independent. Certainly, since this kind of rules are the result of merging several rules whose lhss are variants [2], this distintion corresponds with the intended semantics of the original program.

Example 5. Given the program

$$R_1 : f(X) \to g(X) \qquad R_2 : f(X) \to h(X)$$

it can be transformed in this other program

$$R_{12} : f(X) \to g(X) \mathbin{!} h(X)$$

The variables of the rules R_1 and R_2 of the original program are independent and, indeed, we have to select a variant of these rules when they are applied to perform a computation step. Therefore, it is meaningless to consider the variable X of the rule R_{12} as the same variable for both alternatives.

On the other hand, the situation illustrated by Example 5 does not lead to the unsoundness problem as we have been discussing above. Although the rule R_{12} of Example 5 is not right linear, at most one of both alternatives of the function f is needed in a computation to a value. Therefore, we assume that the apparition of this kind of rules in the transformed program doesn't break the good properties provided by the right linearity of the original program. Certainly, this kind of rules preserve an important compositional property that we are going to establish in Lemma 5.

In order to prove the soundness of our transformation we need the following auxiliary lemmas. The first lemma points out the existence of a compositional property between the minimal elements of a linear term t, the minimal substitutions w.r.t. a substitution σ and the minimal elements of the term $\sigma(t)$.

Lemma 3. *Let t be a linear term and σ a substitution. Let M be the set of all minimal elements of t and $\bigcup_{i=1}^{n}\{\sigma_i\}$ be the set of all minimal substitutions w.r.t. σ. Then, $\bigcup_{i=1}^{n}\sigma_i(M)$ is the set of all minimal elements of the term $\sigma(t)$.*

Proof. By structural induction on t. We distinguish the following cases:

1. $t \in \mathcal{X}$
   In this case t is a minimal element of itself and, by definition of minimal substitution, each $\sigma_i(x)$, $i \in \{1, \ldots, n\}$ is a minimal element of $\sigma(x)$. Therefore, $\bigcup_{i=1}^{n}\{\sigma_i(x)\}$ is the set of all minimal elements of the term $\sigma(t)$.
2. t is a nullary symbol
   Then, the claim trivially follows, since t is the single minimal element of itself, $\sigma(t) = t$ and $\sigma_i(t) = t$, for all $i \in \{1, \ldots, n\}$.
3. $t = f(t_1, \ldots t_m)$
   Since t is linear, each subterm t_j is also linear. Assume M_j is the set of all minimal elements of t_j, $j \in \{1, \ldots, m\}$. By the inductive hypothesis, $\bigcup_{i=1}^{n}\sigma_i(M_j)$ is the set of all minimal elements of the term $\sigma(t_j)$. Now consider the term $s = f(s_1, \ldots s_m)$ where $s_j \in M_j$, $j \in \{1, \ldots, m\}$. By construction s is one of the minimal elements of t and, by Definition 4 and the inductive hypothesis,

   $$\sigma_i(s) = f(\sigma_i(s_1), \ldots \sigma_i(s_m)) \trianglelefteq f(\sigma(t_1), \ldots \sigma(t_m)) = \sigma(t)$$

   that is, $\sigma_i(s)$ is a minimal element of $\sigma(t)$. Therefore, the claim of this lemma immediately follows.

The intuitive idea behind Lemma 3 is to avoid problems like the one illustrated in the following example:

Example 6. Suppose the non-linear term $t = X + X$ (possibly a rhs of a rule) and a substitution $\sigma = \{X/(0 \,!\, s(0))\}$. Then, t is minimal, $\{\{X/0\}, \{X/s(0)\}\}$ is the set of minimal substitutions w.r.t. σ, but $\{\{X/0\}(t), \{X/s(0)\}(t)\} = \{0 + 0, s(0) + s(0)\}$ is not the set of all minimal elements of $\sigma(t)$.

Lemma 3 can be extended to a product where each factor is linear, although some of these factors may share some variables in common.

Lemma 4. *Let $t = u \,!\, v$ be a product where u and v are linear terms such that $\mathcal{V}ar(u) \cap \mathcal{V}ar(v) \neq \emptyset$. Let U and V be the set of all minimal elements of u and v respectively. Let σ be a substitution and $\bigcup_{i=1}^{n}\{\sigma_i\}$ be the set of all minimal substitutions w.r.t. σ. Then, $(\bigcup_{i=1}^{n}\sigma_i(U)) \cup (\bigcup_{i=1}^{n_i}\sigma_i(V))$ is the set of all minimal elements of the term $\sigma(t)$.*

Proof. Since u is a linear term, by Lemma 3, $M_u^\sigma = \bigcup_{i=1}^{n}\sigma_i(U)$ is the set of all minimal elements of the term $\sigma(u)$ and $M_v^\sigma = \bigcup_{i=1}^{n}\sigma_i(V)$ is the set of all minimal elements of the term $\sigma(v)$. Now, consider a term $s \in U$, by Definition 4,

$$\sigma_i(s) \trianglelefteq \sigma(u) \trianglelefteq \sigma(u) \,!\, \sigma(v) = \sigma(t),$$

and, by Corollary 1, $\sigma_i(s)$ is a minimal element of $\sigma(t)$. Therefore, each element of M_u^σ is a minimal element of $\sigma(t)$. Similarly, each element of M_v^σ is a minimal element of $\sigma(t)$. Hence, $M_u^\sigma \cup M_v^\sigma$ is the set of all minimal elements of the term $\sigma(t)$.

Note that, in Lemma 4, the only requirement for the subterms u and v is linearity and, therefore, they can contain occurrences of the alternative symbol. The following lemma is a generalization of Lemma 4.

Lemma 5. *Let $t = t_1 \,!\, , \ldots , \,!\, t_m$ be a product where each factor t_j, $j \in \{1, \ldots, m\}$, is a linear term such that $\bigcap_{j=1}^{m} \mathcal{V}ar(t_j) \neq \emptyset$. Let M_j be the set of all minimal elements of t_j, $j \in \{1, \ldots, m\}$. Let σ be a substitution and $\bigcup_{i=1}^{n}\{\sigma_i\}$ be the set of all minimal substitutions w.r.t. σ. Then, $\bigcup_{i=1}^{n}\bigcup_{j=1}^{m}\sigma_i(M_j)$ is the set of all minimal elements of the term $\sigma(t)$.*

Proof. By induction on the number of alternative symbols in t and Lemma 4.

Note that our transformation produce, mostly, transformed rules whose rhss fulfills the conditions of Lemma 5. Therefore, the last result can be use to obtain all minimal elements of a rhs $\sigma'(r')$ applying the minimal substitutions σ w.r.t. σ' to the minimal elements r of r'. This is one of the keys to prove Lemma 6. But, we first need the introduction of a new concept.

As we said, Definition 2 lets unspecified the shape of the original program as well as the way the transformation steps are applied. Managing this great freedom in the use of our transformation when we try to prove its soundness can increase unnecessarily the difficulty of the proof without produce, as we are going to reason, a real gain. Therefore, in order to maintain the proof as simple as possible, we first assume that the original program is in a "non-factorized" form and we prove some auxiliary results that are extended to arbitrary programs later.

Definition 6 (non-factorized program). *A* non-factorized *program is a program where the rhs of its rules don't contain occurrences of the alternative symbol.*

That is, a non-factorized program is a program where the variant overlapping rules are in the simplest possible form and none factoring transformation was previously applied.

Example 7. The program $\{bigger \to s(0), bigger \to s(s(0))\}$ is a non-factorized version of bigger whereas $\{bigger \to s(0) \,!\, s(s(0))\}$ or $\{bigger \to s(0 \,!\, s(0))\}$ are not.

It is clear that the non-factorized form of a program is unique, since the factoring transformation does not erase factors and it can be undone following this sequence of steps: i) given a rule $R = l \to r$, where r contains the alternative operator, find the set $\{r_1, \ldots, r_n\}$ of minimal elements of r (w.r.t. embedding relation); ii) replace the rule R by the set of rules $\{l \to r_1, \ldots, l \to r_n\}$.

Now, we are ready to prove the following auxiliary lemmas.

Lemma 6. *Let $\mathcal{R}$ be a non-factorized and right linear program and $\mathcal{R}'$ be a program that factors $\mathcal{R}$. If there exists a reduction step $t' \to u'$ in $\mathcal{R}'$ then, for any minimal element u of u' there exists a minimal element t of t' and a reduction step $t \to u$ in $\mathcal{R}$ or $t = u$.*

Proof. By structural induction on t'. We distinguish the following cases:

1. $t' \in \mathcal{X}$ or t' is a nullary constant symbol
   The claim follows by vacuity, since t' is a normal form which is a minimal element of itself.
2. t' is a nullary function symbol
   In this case, t' is a minimal element of itself and there must be a rule $R' = t' \to u'$ in $\mathcal{R}'$ (the step is performed over the top position Λ using the identity substitution id). By Definition 2 and since $\mathcal{R}$ is a non-factorized program, for each minimal element u of u' there must be a rule $R = t' \to u$ in $\mathcal{R}$. Thus, the claim follows trivially.
3. $t' = t'_1 \,!\, t'_2$
   (a) The step is performed over the top position Λ.
   Then the step is performed with an ALT rule. Assume that $t' \to_{ALT_1} t'_1$ is the performed reduction step and that u is a minimal element of t'_1. Then, by Corollary 1, we can construct the reduction sequence $t' \to_{ALT_1} t'_1 \to^* u \not\to_{ALT}$. Therefore, u is also a minimal element of t' and the claim follows trivially since $t = u$.
   (b) The step is performed over a position $p \neq \Lambda$.
   Assume $p = 1.q$ (i.e., p belongs to t'_1) and the step is $t' \to u'_1 \,!\, t'_2$. Then there exists a step $t'_1 \to u'_1$ in $\mathcal{R}'$ and, by the inductive hypothesis, for any minimal element u of u'_1 there exists a minimal element t of t'_1 and a reduction step $t \to u$ in $\mathcal{R}$. Now, the claim follows by Definition 5 and the transitive property of the embedding relation.
   On the other hand, note that, for each minimal element u of t'_2 the claim follows trivially, since then u is also a minimal element of t', that is, $t = u$.
4. $t' = f(t'_1, \ldots t'_n)$
   (a) The step is performed over the top position Λ.
   In this case there must be a rule $R' = l \to r'$ in $\mathcal{R}'$ such that $l = f(C_1[x_{11}, \ldots, x_{1m_1}], \ldots, C_n[x_{n1}, \ldots, x_{nm_n}])$, where each $C_i[\ ]$ is (a possibly empty) constructor context and $t'_i = C_i[s_{i1}, \ldots, s_{nm_i}]$. Therefore there exists a substitution σ' such that it is possible the reduction step $t' = \sigma'(l) \to \sigma'(r')$. By definition of factoring transformation and since $\mathcal{R}$ is a non-factorized program, for each minimal element r of r' there exists a rule $R = l \to r$ in $\mathcal{R}$. Now consider a minimal substitution σ w.r.t. σ', it is possible the following rewriting step $t = \sigma(l) \to \sigma(r) = u$ in $\mathcal{R}$. By Lemma 5 u is a minimal element of u' and, clearly, t is a minimal element of t'. Also, by Lemma 5 all minimal elements of u' can be obtained applaying the minimal substitutions σ w.r.t. σ' to the minimal elements r of r'.

(b) The step is performed over a position $p \neq \Lambda$.
Assume $p = i.q$ (i.e., p belongs to t'_i) and the step is $t' \to f(t'_1, \ldots, u'_i, \ldots, t'_n)$. Then there exists a step $t'_i \to u'_i$ in $\mathcal{R}'$ and, by the inductive hypothesis, for any minimal element u_i of u'_i there exists a minimal element t_i of t'_i and a reduction step $t_i \to u_i$ in $\mathcal{R}$. Now, the claim follows by Definition 4 and the replacement property of rewriting: for any minimal element t_j of t'_j (with $j \neq i$), $u = f(t_1, \ldots, u_i, \ldots, t_n)$ is a minimal element of u', there exits a step $t = f(t_1, \ldots, t_i, \ldots, t_n) \to f(t_1, \ldots, u_i, \ldots, t_n) = u$ in $\mathcal{R}$ and t is a minimal element of t'.

Lemma 7. *Let $\mathcal{R}$ be a non-factorized and right linear program and $\mathcal{R}'$ be a program that factors $\mathcal{R}$. If there exists a reduction sequence $t' \to^* u'$ in $\mathcal{R}'$ then, for any minimal element u of u' there exists a minimal element t of t' and a reduction sequence $t \to^* u$ in $\mathcal{R}$*

Proof. Immediate, by induction on the number of steps n of the reduction sequence in $\mathcal{R}'$ and Lemma 6.

The following lemma establishes the relation between the reduction sequences leading to a value in both the transformed and the original program.

Lemma 8. *Let $\mathcal{R}$ be a non-factorized and right linear program and $\mathcal{R}'$ be a program that factors $\mathcal{R}$. If there exists a reduction sequence $t \to^* s$ in $\mathcal{R}'$, where s is a value then there exists a reduction sequence $t \to^* s$ in $\mathcal{R}$*

Proof. By Lemma 7 there exists a reduction sequence $u \to^* v$ in $\mathcal{R}$, where u is a minimal element of t and v a minimal element of s. Since $u \trianglelefteq t$, by Lemma 1, there exists a reduction sequence $t \to_{ALT} \ldots \to_{ALT} u$. On the other hand, if s is a value and $v \trianglelefteq s$, then $v = s$. Therefore, we can construct the following reduction sequence in $\mathcal{R}$: $t \to_{ALT} \ldots \to_{ALT} u \to^* v = s$.

Now, we lift the last result to arbitrary right linear programs.

Proposition 2 (pre-soundness). *Let $\mathcal{R}$ be a right linear program and $\mathcal{R}'$ be a program that factors $\mathcal{R}$. If there exists a reduction sequence $t \to^* s$ in $\mathcal{R}'$, where s is a value then there exists a reduction sequence $t \to^* s$ in $\mathcal{R}$*

Proof. Assume $\mathcal{R}''$ is the non-factorized program such that $\mathcal{R}$ factors $\mathcal{R}''$. By definition of the factoring transformation it is clear that $\mathcal{R}''$ exists and $\mathcal{R}'$ factors $\mathcal{R}''$. Also note that, being $\mathcal{R}$ right linear, $\mathcal{R}''$ must be right linear. Now, by Lemma 8, if there exists a reduction sequence $t \to^* s$ in $\mathcal{R}'$, then there exists a reduction sequence $t \to^* s$ in $\mathcal{R}''$ and therefore, by Proposition 1, there exists a reduction sequence $t \to^* s$ in $\mathcal{R}$.

The soundness of the transformation is an easy consequence of the correctness of narrowing and Proposition 2

Theorem 2. *The factoring transformation is sound for right linear programs.*

Proof. Let $\mathcal{R}$ be a right linear program and $\mathcal{R}'$ be a program that factors $\mathcal{R}$. If there exists a derivation $t \leadsto_\sigma^* s$ in $\mathcal{R}'$

$$\begin{array}{ll} \Rightarrow \sigma'(t) \to^* s \text{ in } \mathcal{R}' & \text{(by soundness of narrowing)} \\ \Rightarrow \sigma'(t) \to^* s \text{ in } \mathcal{R} & \text{(by Proposition 2)} \\ \Rightarrow t \leadsto_\sigma^* s \text{ in } \mathcal{R} \text{ with } \sigma = \sigma'[\mathcal{V}ar(t)] & \text{(by completeness of narrowing).} \end{array}$$

4.4 Discussion

As we have seen, factoring is generally unsound, but soundness can be recovered in some cases of practical interest. In the following discussion, the notions of *descendant* of a redex is informal, as this notion has been rigorously defined only for orthogonal rewrite systems [12]. The unsoundness of factoring originates from computations in which some redex r generates several descendants, and distinct descendants are reduced to different terms. Thus, two simple independent solutions to the unsoundness problem are to avoid distinct descendants of a redex or to ensure that all the descendants of a redex are contracted using the same rewrite rule.

The first condition holds, as we have just proved, for right-linear rewrite systems. But the restriction to right linear systems is unacceptable from a programming point of view.

The second condition is ensured by the use of a call-time choice semantics or sharing. These solutions were proposed to manage similar but distinct issues in the context of overlapping TRSs by Hussmann [14]. Informally, the *call-time choice* semantics consists of "committing" to the value of an argument of a symbol (either operation or constructor) at the time of the symbol's application. The value of the argument does not need to be computed at application time: thus, the laziness of computations is not jeopardized. This semantics is the most appropriate for some computations, but it fails to capture the intuitive meaning of others. Nevertheless, there are formalisms, such as [9], and languages, like Toy [17] (based on the aforementioned formalism) and Curry [11], that adopt the call-time choice as the only semantics of non-deterministic computations. On the other hand, *sharing* is a technique where all occurrences of the same variable in a rhs of a rule are shared, i.e. all these occurrences are replaced by a pointer to the same subterm after a rewriting step. Sharing is effectively implemented using term graphs.

Hence, both approaches, call-time choice semantics or sharing, are useful to resolve the problems discussed w.r.t. Example 4: they avoid the strong syntactic restriction of right linearity while keep the factoring transformation sound.

5 Conclusions

Non-deterministic computations are an essential feature of functional logic programming languages but its implementation may be very costly, since, usually it is necessary to compute a set of fair independent computations whose results are used, and possibly discarded, by a context.

In this paper, we have defined a program transformation, called factoring transformation, that can be fully automated or used as a programming technique. The factoring transformation is based on the introduction of a new symbol, called *alternative*, into the signature of a program. The alternative symbol is a polymorphic defined operation. This symbol allows a program to factor a common portion of the non-deterministic replacements of a redex. This transformation may improve the efficiency of a computation by reducing the number of computation steps or the memory used in representing terms. Savings are obtained when fair independent computations are avoided because only the factored portion of non-deterministic replacements is needed.

We have studied the formal properties of the factoring transformation, proving its correctness for right linear programs. Afterwards, we have discussed how this impractical syntactic restriction can be overcome if sharing or call-time choice semantics is used as an implementation device of the functional logic language. Therefore, our transformation can be applied safely to the Constructor-based (conditional) ReWriting Logic (CRWL) programs of [9] as well as to Curry programs [11], since they adopt the call-time choice semantics for non-deterministic functions.

Also, in order to prove our results we have introduced an embedding relation that have shown its usefulness to reason with non-confluent constructor-based TRSs, where the notion of descendants of a redex [12] is not well established.

Finally note that, although the process of factoring a program can remind, in some aspects, the full laziness transformation of Functional Programming [13, 15], at the best of our knowledge, it is a novel transformation that can be applied to a high programming language as well as to a core language, producing an effective improvement of the efficiency of the programs.

Acknowledgments

I wish to thank Sergio Antoy the discussion of technical aspects of this paper that greatly improve it. Part of this research was done while the author was visiting the Department of Computer Science at Portland State University. The author gratefully acknowledges the hospitality of that department.

References

1. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of PEPM'97*, volume 32, 12 of *Sigplan Notices*, pages 151–162, New York, 1997. ACM Press.
2. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of ALP'97*, pages 16–30. Springer LNCS 1298, 1997.
3. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, July 2000.
4. S. Antoy, P. Julián Iranzo, and B. Massey. Improving the efficiency of non-deterministic computations. *Electronic Notes in Theoretical Computer Science*, 64:22, 2002. URL: `http://www.elsevier.nl/locate/entcs/volume64.html`.

5. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
6. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
7. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.
8. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. ESOP'96*, pages 156–172. Springer LNCS 1058, 1996.
9. 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.
10. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
11. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Ver. 0.71). Web document `http://www.informatik.uni-kiel.de/~mh/curry/report.html`, 2000.
12. 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.
13. R. Hughes. *The Design and Implementation of Programming Languages.* PhD thesis, University of Oxford, 1984., 1984.
14. H. Hussmann. Nondeterministic Algebraic Specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
15. S. L. Peyton Jones and D. Lester. A modular fully-lazy lambda lifter in HASKELL. *Software, Practice and Experience*, 21(5):479–506, 1991.
16. M. Leuschel and B. Martens. Global Control for Partial Deduction through Characteristic Atoms and Global Trees. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, pages 263–283. Springer LNCS 1110, 1996.
17. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proceedings of RTA '99*, pages 244–247. Springer LNCS 1631, 1999.
18. A. Pettorossi and M. Proietti. A Comparative Revisitation of Some Program Transformation Techniques. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 355–385. Springer LNCS 1110, 1996.