Redundancy Analyses in Term Rewriting (extended abstract)*

María Alpuente Santiago Escobar Salvador Lucas

DSIC, UPV, Camino de Vera s/n, 46022 Valencia, Spain.
{alpuente,sescobar,slucas}@dsic.upv.es

Abstract. The uncontrolled application of automatic transformation processes during the formal development and optimization of programs can introduce encumbrances in the generated code that programmers usually (or presumably) do not write. An example is the introduction of redundant arguments in the functions defined in the program. Redundancy of a parameter means that replacing it by any value does not change the result since the value is not used. In this work, we give a theoretical foundation for the analysis and elimination of redundant arguments in term rewriting systems as a model for the programs that can be written in more sophisticated declarative languages. The framework is independent from both the considered semantics and the framework - operational or denotational- which is used to describe (and approximate) the observed meaning. We particularize our method to the standard (reduction) semantics of term rewriting systems including the standard normalization semantics (typical of pure rewriting) and the evaluation semantics (closer to functional programming). On the basis of the analyses, we formalize an erasure procedure which preserves the semantics and produces smaller programs which are easier to understand and maintain. Finally, we indicate how the present method can be extended to more sophisticated program observable properties such as computed answers.

Keywords: Term rewriting, declarative programming, formal methods, semantics, program analysis and optimization

1 Introduction

The formal development and optimization of programs involve the use of systematic transformations which preserve the abstract semantics of the original (specification of a) program while typically producing a more efficient version (or implementation). It is desirable that these tranformations be as automated as possible, i.e., with no explicit intervention of the user (programmer or implementor). Partial evaluation techniques are a good example of transformations which can achieve these goals. Partial evaluation (PE) is a semantics-preserving performance optimization technique for computer programs which consists of the specialization of the program w.r.t. parts of its input. PE has been widely

* This work has been partially supported by CICYT TIC 98-0445-C03-01 and by Acción Integrada hispano-alemana HA1997-0073.

applied in term rewriting [Bel95,Bon88,DR93,MS96], functional programming [CD93,JGS93], logic programming [Gal93,LS91], and integrated functional-logic languages [AFJV97,AFV98a,AFV98b,AHLV99,LG97].

A number of researchers have noticed that certain processes of optimization, transformation, specialization and reuse of code often introduce anomalies in the generated code that programmers usually (or ideally) do not write [ASU86,Hug88,LS96]. Examples are redundant arguments in the functions defined by the program, as well as useless clauses or program rules. For example, consider the following functions:

```
last(x:nil)    → x                append(nil,y)    → y
last(x:y:ys)   → last(y:ys)       append(x:xs,y)   → x:append(xs,y)
```

and the call `applast(ys,z)` $\equiv$ `last(append(ys,z:nil))` which appends an element `z` at the end of a given list `ys` and then returns the last element of the resulting list (the example is borrowed from DPPD library of benchmarks [Leu98]). Assume that we specialize the program for this call. Commonly, the specialized program which can be obtained (with no further optimization) by using an automatic partial evaluator of functional programs such as the one described in [AFJV97,AFV98a] is:

```
applast(x:xs,z) → lastnew(x,xs,z)
applast(nil,z)  → z
lastnew(x,y:ys,z) → lastnew(y,ys,z)
lastnew(x,nil,z)  → z
```

This is too far from {`applast(ys,z)` $\to$ `lastnew′(z)`, `lastnew′(z)` $\to$ `z`}, a more feasible program, or even the "optimal" program {`applast(ys,z)` $\to$ `z`} which one would ideally expect. Indeed, note that the first argument of the function `applast` is redundant (as well as the first and the second parameters of the auxiliary function `lastnew`) and would not typically be written by a programmer who writes this program by hand. Also note that the standard (post-specialization) renaming/compression procedures cannot perform this optimization as they only improve programs where program calls contain dead functors or multiple occurrences of the same variable, or the functions are defined by rules whose rhs's are normalizable [AFJV97,Gal93,GS94].

It seems interesting to formalize program analysis techniques for detecting these kinds of redundancies as well as to formalize transformations for eliminating dead code which appears in the form of redundant function arguments or useless rules and which, in some cases, can be safely erased without jeopardizing correctness. In this work, we investigate the problem of redundant arguments in term rewriting systems, as a model for the programs that can be written in more sophisticated multiparadigm languages. We provide a semantic definition of redundancy which is parametric w.r.t. the observed semantics. We consider different (reduction) semantics including the standard normalization semantics (typical of pure rewriting) and the evaluation semantics (closer to functional programming). The problem of how to recognize redundancy of arguments under different semantics is simplified to a (weaker) notion of local redundancy which can be recognized statically by inspecting program rules. We formalize some methods for analyzing local redundancy which are the basis for a correct approximation of redundancy. Finally, we formalize an elimination procedure which gets rid of the useless arguments and preserves the semantics of the original program. We conclude with some comparisons with the related literature and future work. Proofs of theorems are given in an appendix.

2 Preliminaries

Let us first introduce the main notations used in the paper. For full definitions we refer to [BN98]. Given a set A, $|A|$ denotes its cardinality, and $\mathcal{P}(A)$ denotes the set of all subsets of A. Let $R \subseteq A \times A$ be a binary relation on a set A. We denote the transitive closure of R by R^+ and its reflexive and transitive closure by R^*. We say that R is *confluent* if, for every $a, b, c \in A$, whenever $a\ R^* b$ and $a\ R^* c$, there exists $d \in A$ such that $b\ R^* d$ and $c\ R^* d$. We say that R is *terminating* iff there is no infinite sequence $a_1\ R\ a_2\ R\ a_3 \cdots$.

Throughout the paper, $\mathcal{X}$ denotes a countable set of variables and Σ denotes a set of function symbols $\{\mathtt{f}, \mathtt{g}, \ldots\}$, each one having a fixed arity given by a function $ar : \Sigma \to \mathbb{N}$; sometimes we write f/k if $ar(f) = k$. If $ar(f) = 0$, we say that f is a constant symbol. By $\mathcal{T}(\Sigma, \mathcal{X})$ we denote the set of terms; $\mathcal{T}(\Sigma)$ is the set of ground terms, i.e., terms without variable occurrences. A term is said to be linear if it has no multiple occurrences of a single variable. A k-tuple $t_1, \ldots, t_k$ of terms is written $\overline{t}$. The number k of elements of the tuple $\overline{t}$ will be clarified by the context. $\mathcal{V}ar(t)$ is the set of variables in t. A *substitution* is a mapping $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$ which homomorphically extends to a mapping $\sigma : \mathcal{T}(\Sigma, \mathcal{X}) \to \mathcal{T}(\Sigma, \mathcal{X})$. Let $Subst(\Sigma, \mathcal{X})$ denote the set of substitutions and $Subst(\Sigma)$ be the set of ground substitutions, i.e., substitutions on $\mathcal{T}(\Sigma)$. We denote by id the "identity" substitution: $id(x) = x$ for all $x \in \mathcal{X}$. The set $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is called the *domain* of σ and $\mathcal{R}ng(\sigma) = \cup_{x \in \mathcal{D}om(\sigma)} \mathcal{V}ar(\sigma(x))$ its *range*. A *unifier* of two terms t, s is a substitution σ with $\sigma(t) = \sigma(s)$. A *most general unifier* (*mgu*) of t, s is a unifier σ such that for each unifier σ' of t, s there exists θ such that $\sigma' = \theta \circ \sigma$.

Terms are viewed as labelled trees in the usual way. Positions $p, q, \ldots$ are represented by chains of positive natural numbers used to address subterms of t. By Λ, we denote the empty chain. If p is a position, and Q is a set of positions, $p.Q$ is the set $\{p.q \mid q \in Q\}$. Positions are ordered by the standard prefix ordering: $p \leq q$ iff $\exists q'$ such that $q = p.q'$; as usual, $p < q$ means that $p \leq q$ and $p \neq q$. The subterm at position p of t is denoted as $t|_p$ and $t[s]_p$ is the term t with the subterm at position p replaced by s. By $\mathcal{P}os(t)$ we denote the set of positions of a term t. The symbol labelling the root of t is denoted as $root(t)$. A context is a term C with zero or more 'holes', $\Box$ (a fresh constant symbol). We write $C[\ ]_p$ to denote that there is a (usually single) hole $\Box$ at position p of C. We usually write simply $C[\ ]$ to denote arbitrary context, clarifying the number and location of holes 'in situ'. $C[\ ] = \Box$ is called the *empty* context. If C is a context and t a term, $C[t]$ denotes the result of replacing the hole in C by t.

A rewrite rule is an ordered pair (l, r), written $l \to r$, with $l, r \in \mathcal{T}(\Sigma, \mathcal{X})$, $l \notin \mathcal{X}$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The left-hand side (*lhs*) of the rule is l and r is the right-hand side (*rhs*). A TRS is a pair $\mathcal{R} = (\Sigma, R)$ where R is a set of rewrite rules. $L(\mathcal{R})$ denotes the set of *lhs*'s of $\mathcal{R}$. An instance $\sigma(l)$ of a *lhs* l of a rule is a redex. A term t without redexes is said a normal form. By $\mathsf{NF}_{\mathcal{R}}$ we denote the set of finite normal forms of $\mathcal{R}$. A TRS $\mathcal{R}$ is left-linear if for all $l \in L(\mathcal{R})$, l is a linear term. Given $\mathcal{R} = (\Sigma, R)$, we consider Σ as the disjoint union $\Sigma = \mathcal{C} \uplus \mathcal{F}$ of symbols $c \in \mathcal{C}$, called *constructors* and symbols $f \in \mathcal{F}$, called *defined functions*, where $\mathcal{F} = \{f \mid f(\bar{l}) \to r \in R\}$ and $\mathcal{C} = \Sigma - \mathcal{F}$. Then, $\mathcal{T}(\mathcal{C}, \mathcal{X})$ is the set of constructor terms. A pattern is a term $f(l_1, \ldots, l_n)$ such that $f \in \mathcal{F}$ and $l_1, \ldots, l_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. The set of patterns is $\mathcal{P}att(\Sigma, \mathcal{X})$. A term t rewrites to s (at position p), written $t \to_{\mathcal{R}} s$ (or just $t \to s$), if $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$, for some rule $l \to r \in R$, $p \in \mathcal{P}os(t)$ and substitution σ. This can eventually be detailed by writing $t \stackrel{p}{\to} s$. The one-step rewrite relation for $\mathcal{R}$ is $\to$. A term t is root-stable if it cannot be rewritten to a redex.

The set of infinite terms is denoted by $\mathcal{T}^{\infty}(\Sigma, \mathcal{X})$ (see [KKSV95] for a formal definition of infinite term). An infinite rewrite sequence is an infinite sequence $t_1, t_2, \ldots$ of terms such that $t_n \to t_{n+1}$ for all $n \in \mathbb{N}$. Following [Mid97], in this paper we are only interested in infinite rewrite sequences of length ω (unlike [KKSV95]). We write $t \to^{\omega} s$ if either $t \to^* s$ or s is the (possibly infinite) limit of an infinite rewrite sequence starting from t (according to the standard notion of Cauchy convergence, see [KKSV95] for details). If $t \to^{\omega} s$, then s is a reduct of t. $\mathsf{NF}^{\omega}_{\mathcal{R}}$ denotes the set of finite or infinite normal forms of $\mathcal{R}$.

3 Rewriting semantics

The redundancy of an argument of a function f in a TRS $\mathcal{R}$ depends on the semantic properties of $\mathcal{R}$ that we are interested in observing.

Definition 1. *A* rewriting semantics *for a TRS $\mathcal{R}$ is a function $\mathsf{S} : \mathcal{T}(\Sigma, \mathcal{X}) \to \mathcal{P}(\mathcal{T}^{\infty}(\Sigma, \mathcal{X}))$ such that, for all $t \in \mathcal{T}(\Sigma, \mathcal{X})$, $\mathsf{S}(t)$ is a set of reducts of t.*

By a *finitary* rewriting semantics we mean a semantics S that only consider reducts reached by means of a finite number of rewriting steps, i.e., $\forall t \in \mathcal{T}(\Sigma, \mathcal{X})$, $s \in \mathsf{S}(t) \Rightarrow t \to^* s$. In this paper, we restrict the discussion of rewriting semantics to *ground* rewriting semantics, i.e., functions $\mathsf{S} : \mathcal{T}(\Sigma) \to \mathcal{P}(\mathcal{T}^{\infty}(\Sigma))$. The (finitary) semantics which is most commonly considered in functional programming is the set of values (finite normal forms which are ground constructor terms) that $\mathcal{R}$ is able to produce in a finite number of rewriting steps (eval). Other kinds of finitary semantics often considered for $\mathcal{R}$ are, e.g., the set of all possible reducts of a term which are reached in a finite number of steps (red), the set of such reducts that are ground head-normal forms (i.e., ground root-stable terms) (hnf), or ground normal forms (nf). We also have the infinitary counterparts: the set of all (possibly infinite) reducts ($\omega{-}\mathsf{red}$), reducts that are ground head-normal forms ($\omega{-}\mathsf{hnf}$), ground normal forms ($\omega{-}\mathsf{nf}$), or ground values ($\omega{-}\mathsf{eval}$). We

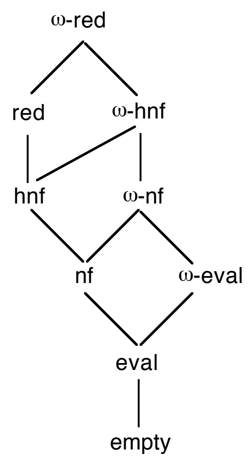

Fig. 1. Semantics for TRSs ordered by $\preceq$ ($\sqsubseteq$)

also consider the (trivial) semantics empty which assigns an empty set to every term. These semantics are related: for every TRS $\mathcal{R}$ and term $t \in \mathcal{T}(\Sigma)$,

$$\begin{aligned} \mathsf{red}(t) &= \omega{-}\mathsf{red}(t) \cap \mathcal{T}(\Sigma) & \omega{-}\mathsf{nf}(t) &= \omega{-}\mathsf{hnf}(t) \cap \mathsf{NF}^{\omega}_{\mathcal{R}} \\ \mathsf{hnf}(t) &= \omega{-}\mathsf{hnf}(t) \cap \mathcal{T}(\Sigma) & \omega{-}\mathsf{eval}(t) &= \omega{-}\mathsf{nf}(t) \cap \mathcal{T}^{\infty}(\mathcal{C}) \\ \mathsf{nf}(t) &= \omega{-}\mathsf{nf}(t) \cap \mathcal{T}(\Sigma) & \mathsf{eval}(t) &= \omega{-}\mathsf{eval}(t) \cap \mathcal{T}(\Sigma) \\ \mathsf{empty}(t) &= \varnothing & &= \mathsf{nf}(t) \cap \mathcal{T}(\mathcal{C}) \end{aligned}$$

We say that a semantics S for a TRS $\mathcal{R}$ is $\mathcal{R}$-normalizing (or $(\mathcal{R},\omega)$-normalizing) if for all $t \in \mathcal{T}(\Sigma)$, $\mathsf{S}(t) \subseteq \mathsf{NF}_{\mathcal{R}}$ ($\mathsf{S}(t) \subseteq \mathsf{NF}^{\omega}_{\mathcal{R}}$). We say that S is a forwarding semantics if for all $t \notin \mathsf{NF}_{\mathcal{R}}$, $\mathsf{S}(t) = \cup_{t \to^{+} t'} \mathsf{S}(t')$. Note that forwarding semantics are $\mathcal{R}$-normalizing; $\mathsf{eval}, \mathsf{nf}, \omega{-}\mathsf{eval}$ and $\omega{-}\mathsf{nf}$ are examples of forwarding semantics (hence, they are $\mathcal{R}$-normalizing too). On the other hand, $\mathsf{hnf}, \mathsf{red}, \omega{-}\mathsf{hnf}$, and $\omega{-}\mathsf{red}$ are neither forwarding nor $\mathcal{R}$-normalizing.

Given semantics S, S' for $\mathcal{R}$, we write $\mathsf{S} \preceq \mathsf{S}'$ if there exists $T \subseteq \mathcal{T}^{\infty}(\Sigma)$ such that, for all $t \in \mathcal{T}(\Sigma)$, $\mathsf{S}(t) = \mathsf{S}'(t) \cap T$. We have that $\preceq$ is an ordering (see Figure 1). Note that empty is the most restrictive semantics. We can also trivially compare two semantics S and S' for a TRS $\mathcal{R}$ by means of an ordering $\sqsubseteq$ as follows: $\mathsf{S} \sqsubseteq \mathsf{S}'$ if for all $t \in \mathcal{T}(\Sigma)$, $\mathsf{S}(t) \subseteq \mathsf{S}'(t)$, i.e., S is a restriction of S'. Note that $\mathsf{S} \preceq \mathsf{S}'$ implies $\mathsf{S} \sqsubseteq \mathsf{S}'$, but not vice versa.

Example 1. Consider the following TRS:

```
f(x) → a          b → b
```

and the semantics $\mathsf{nf_{in}}$ such that $\mathsf{nf_{in}}(t)$ is the set of normal forms that are obtained by means of innermost reductions of t (i.e., by contracting redexes that do not contain other redexes). Note that $\mathsf{nf_{in}} \sqsubseteq \mathsf{nf}$. However, $\mathsf{nf_{in}} \not\preceq \mathsf{nf}$: we have that $\mathsf{nf_{in}}(\mathtt{a}) = \mathsf{nf}(\mathtt{a}) = \{\mathtt{a}\}$. Thus, if there is T such that for all $t \in \mathcal{T}(\Sigma)$, $\mathsf{nf_{in}}(t) = \mathsf{nf}(t) \cap T$, it must happen that $\mathtt{a} \in T$. However, $\mathsf{nf_{in}}(\mathtt{f(b)}) = \varnothing$ and $\mathsf{nf}(\mathtt{f(b)}) = \{\mathtt{a}\}$. Hence, $\mathsf{nf}(\mathtt{f(b)}) \cap T = \{\mathtt{a}\} \cap T = \{\mathtt{a}\} \neq \varnothing = \mathsf{nf_{in}}(\mathtt{f(b)})$.

The ordering $\preceq$ provides some interesting properties regarding the redundancy of arguments w.r.t. those semantics which are comparable by $\preceq$ and which do not hold for the ordering $\sqsubseteq$ as we will show in the following section.

3.1 S-Completeness

Given a TRS $\mathcal{R}$ and a semantic description S for $\mathcal{R}$, it is interesting to investigate whether S always provides non-trivial information about computations issued by $\mathcal{R}$.

Definition 2. *Let $\mathcal{R}$ be a TRS and S be a semantics for $\mathcal{R}$. We say that $\mathcal{R}$ is* S*-complete if for all $t \in \mathcal{T}(\Sigma)$, $\mathsf{S}(t) \neq \varnothing$.*

Note that S-completeness is not only a property of TRSs but also of semantics. For instance, since $\mathsf{empty}(t) = \varnothing$ for all $t \in \mathcal{T}(\Sigma)$, there is no empty-complete TRS. On the other hand, since $t \in \mathsf{red}(t)$ for all $t \in \mathcal{T}(\Sigma)$, every TRS is red-complete. In general, if $\mathsf{S} \sqsubseteq \mathsf{S}'$ (or $\mathsf{S} \preceq \mathsf{S}'$) and $\mathcal{R}$ is S-complete, it is also S'-complete.

S-completeness plays a prominent role in our development. Thus, it is interesting to be able to analyze whether a TRS is S-complete. Completeness of rewriting computations with respect to different semantics have already been studied in the literature. For instance, nf-complete TRSs are usually known as *weakly normalizing* TRSs [BN98]. Terminating TRSs are weakly normalizing; hence, proofs of termination of TRSs (see [Der87] for a survey on this topic) can be used for ensuring nf-completeness of TRSs.

The eval-completeness is related to the standard notion of *completely defined* (CD) TRS's (e.g., see [Han94,HH80]). A function symbol is completely defined if it does not occur in any ground term in normal form. A TRS $\mathcal{R}$ is completely defined if each function symbol of the signature is completely defined. The following proposition is an straight consequence of the results in [HH80].

Proposition 1. *Let $\mathcal{R}$ be a terminating TRS. Then, $\mathcal{R}$ is completely defined iff $\mathcal{R}$ is* eval*-complete.*

In [HH80], a sufficient, syntactical criterion is given to decide whether a terminating and finite TRS $\mathcal{R}$ is completely defined, hence eval-complete. Roughly speaking, for every $f \in \Sigma$ there is in $\mathcal{R}$ a set of rules whose lhs's are of the form $f(s_1^i, \ldots, s_k^i)$, $1 \leq i \leq p$, and the set $S = \{S^1, \ldots, S^p\}$ of k-tuples $\langle s_1^i, \ldots, s_k^i \rangle$ is a *basis* for C, that is, S 'covers' every possible tuple of constructor terms. This is to say that, for every k-tuple $\langle t_1, \ldots, t_k \rangle$ of ground constructor terms, there exists q, with $1 \leq q \leq p$, and a substitution $\sigma \in \mathit{Subst}(\Sigma, \mathcal{X})$ s.t. for every ℓ, $1 \leq \ell \leq k$, we have $t_\ell = \sigma(s_\ell^q)$.

Example 2. Consider the following TRS $\mathcal{R}$:

$$\texttt{zero(0)} \to \texttt{0} \qquad \texttt{zero(1)} \to \texttt{0} \qquad \texttt{zero(zero(x))} \to \texttt{0}$$

If the signature does not contain any symbol not occurring in the TRS, $\mathcal{R}$ is eval-complete (actually, the third rule is not necessary for eval-completeness but engaging for making more general Examples 10 and 12).

4 Redundant arguments in rewriting

Roughly speaking, a redundant argument of a function f is an argument t_i which we do not need to consider in order to compute the semantics of any call containing a subterm $f(t_1, \ldots, t_k)$.

Definition 3 (Redundancy of an argument). *Let $\mathcal{R}$ be a TRS,* S *be a semantics for $\mathcal{R}$, $f \in \Sigma$, and $i \in \{1, \ldots, ar(f)\}$. The i-th argument of f is* redundant w.r.t. $\mathcal{R}$ and S *if, for all context $C[\,]$ and for all $t, s \in \mathcal{T}(\Sigma)$ such that $root(t) = f$, $\mathsf{S}(C[t]) = \mathsf{S}(C[t[s]_i])$.*

We denote by $rarg_{\mathcal{R},\mathsf{S}}(f)$ the set of redundant arguments of a symbol f w.r.t. a TRS $\mathcal{R}$ and a semantics S. Note that the semantics empty provides an example of redundancy since, given a TRS $\mathcal{R}$, every argument of every symbol is redundant w.r.t. $\mathcal{R}$ and empty. However, the following example illustrates that there can be redundancies even w.r.t. the greatest semantics in the hierarchy depicted in Figure 1 (i.e., $\omega{-}\mathsf{red}$).

Example 3. Consider the following TRS $\mathcal{R}$:

```
f(x) → x
f(x) → f(f(a))
```

For all $m > 0$, $\omega{-}\mathsf{red}(\mathtt{f}^m\mathtt{(a)}) = \{\mathtt{f}^n\mathtt{(a)} \mid n \leq \omega\}$. Thus, since ground terms are of the form $\mathtt{f}^m\mathtt{(a)}$ with $m \geq 0$, it follows that for $t = \mathtt{f}^m\mathtt{(a)}$, $s = \mathtt{f}^{m'}\mathtt{(a)}$, and context $C[\,] = \mathtt{f}^k(\Box)$, $\omega{-}\mathsf{red}(C[\mathtt{f}(t)]) = \{\mathtt{f}^n\mathtt{(a)} \mid n \leq \omega\} = \omega{-}\mathsf{red}(C[\mathtt{f}(s)])$, i.e., $1 \in rarg_{\mathcal{R},\omega-\mathsf{red}}(\mathtt{f})$.

Redundancy is antimonotone with regard to the ordering $\preceq$ on semantics.

Theorem 1. *Let $\mathcal{R}$ be a TRS and S, S' be semantics for $\mathcal{R}$. If $\mathsf{S} \preceq \mathsf{S}'$, then, for all $f \in \Sigma$, $rarg_{\mathcal{R},\mathsf{S}'}(f) \subseteq rarg_{\mathcal{R},\mathsf{S}}(f)$.*

Note that a similar result does not hold for the ordering $\sqsubseteq$, as witnessed by the following.

Example 4. Consider the TRS $\mathcal{R}$ of Example 1. The argument of $\mathtt{f}$ is clearly redundant w.r.t. nf. However, it is *not* redundant w.r.t. $\mathsf{nf_{in}}$: $\mathsf{nf_{in}}(\mathtt{f(b)}) = \varnothing \neq \{\mathtt{a}\} = \mathsf{nf_{in}}(\mathtt{f(a)})$.

When analyzing some property of a function f in $\mathcal{R}$, it is useful to get rid of the contexts and perform easier, local analyses which allow us to center the attention on the syntactic structure of the rewriting rules. This motivates the following.

Definition 4 (Local redundancy of an argument). *Let $\mathcal{R}$ be a TRS,* S *be a semantics for $\mathcal{R}$, $f \in \Sigma$, and $i \in \{1, \ldots, ar(f)\}$. The i-th argument of f is* locally redundant w.r.t. $\mathcal{R}$ and S *if, for all $t, s \in \mathcal{T}(\Sigma)$ such that $root(t) = f$, $\mathsf{S}(t) = \mathsf{S}(t[s]_i)$.*

We denote by $lrarg_{\mathcal{R},\mathsf{S}}(f)$ the set of locally redundant arguments of a symbol f w.r.t. $\mathcal{R}$ and S.

Theorem 2. *Let $\mathcal{R}$ be a TRS and S, S' be two semantics for $\mathcal{R}$. If $\mathsf{S} \preceq \mathsf{S}'$, then, for all $f \in \mathcal{F}$, $lrarg_{\mathcal{R},\mathsf{S}'}(f) \subseteq lrarg_{\mathcal{R},\mathsf{S}}(f)$.*

Proposition 2. *Let $\mathcal{R}$ be a TRS and $c \in \mathcal{C}$. Then, $lrarg_{\mathcal{R},\mathsf{eval}}(c) = \varnothing$.*

Proposition 2 and Theorem 2 entail the following.

Corollary 1. *Let $\mathcal{R}$ be a TRS and S be such that $\mathsf{eval} \preceq \mathsf{S}$. For all $c \in \mathcal{C}$, $lrarg_{\mathcal{R},\mathsf{S}}(c) = \varnothing$ and $rarg_{\mathcal{R},\mathsf{S}}(c) = \varnothing$.*

Hence, Corollary 1 means that, w.r.t. the 'interesting' semantics shown in Figure 1, we can restrict our attention to defined symbols.

4.1 Using local redundancy for characterizing redundancy

Redundancy of an argument w.r.t. a TRS $\mathcal{R}$ and a semantics S implies local redundancy w.r.t. $\mathcal{R}$ and S, i.e., $rarg_{\mathcal{R},\mathsf{S}}(f) \subseteq lrarg_{\mathcal{R},\mathsf{S}}(f)$. Unfortunately, the converse statement is not generally true.

Example 5. Let $\mathcal{R}$ be the following TRS:

```
from(n) → n:from(s(n))        head(x:xs) → x
```

Assume that the constant constructor `0` belongs to the signature. According to Definition 4, the argument of `from` is locally redundant w.r.t. $\mathcal{R}$ and eval, since there is no derivation issuing from a term rooted by `from` which could produce a value. However, the argument of the function `from` is not redundant w.r.t. $\mathcal{R}$ and eval since the values of the calls `head(from(0))` and `head(from(s(0)))` are different using the context `head(□)`.

Also note that the argument of `from` is not locally redundant (nor redundant) w.r.t. $\omega\mathsf{-eval}$ since, e.g., the infinite values `0:s(0):`$\cdots$ and `s(0):s(s(0)):`$\cdots$ of the calls `from(0)` and `from(s(0))`, respectively, differ.

Finally, note that $\mathcal{R}$ is not eval-complete ($\omega\mathsf{-eval}$-complete).

The above example suggests that S-completeness is a critical condition for redundancy to boil down to local redundancy. The following example reveals that the additional requirement for $\mathcal{R}$-normalization must also be satisfied.

Example 6. Consider the following (confluent) TRS $\mathcal{R}$:

```
f(g(0)) → f(0)        g(x) → 0
```

and the semantics red^{Λ} defined by: $\mathsf{red}^{\Lambda}(t) = \{s \mid t \overset{\Lambda}{\to} s\}$ if t is a redex, and $\mathsf{red}^{\Lambda}(t) = \{t\}$, otherwise. Note that $\mathcal{R}$ is trivially red^{Λ}-complete; however, red^{Λ} is not $\mathcal{R}$-normalizing. Now, the argument of `g` is locally redundant w.r.t. $\mathcal{R}$ and red^{Λ}, whereas it is not redundant, since `f(0)` $\in \mathsf{red}^{\Lambda}($`f(g(0))`$)$ but `f(0)` $\notin \mathsf{red}^{\Lambda}($`f(g(g(0)))`$)$.

Theorem 3. *Let $\mathcal{R}$ be a confluent TRS and S be an $\mathcal{R}$-normalizing finitary semantics for $\mathcal{R}$. If $\mathcal{R}$ is S-complete, then, for all $f \in \Sigma$, $lrarg_{\mathcal{R},\mathsf{S}}(f) = rarg_{\mathcal{R},\mathsf{S}}(f)$.*

By Theorem 3, in a confluent, terminating and CD TRS $\mathcal{R}$, redundancy and local redundancy w.r.t. eval coincide since $\mathcal{R}$ is eval-complete (by Proposition 1) and eval is $\mathcal{R}$-normalizing.

Note that $\mathcal{R}$ in Example 5 is not nf-complete (hence, eval-complete) since no term from(t) has a normal form. Confluence cannot be droped in Theorem 3.

Example 7. Let $\mathcal{R}$ be the (non-confluent) terminating TRS

```
zero(0)  → 0          zero(start(x)) → 0
zero(1)  → 0          start(zero(0)) → 0
```

Since $\mathcal{R}$ is terminating, it is nf-complete. According to Definition 4, the argument of the function zero is locally redundant w.r.t. nf, since any derivation issuing from a term rooted by zero evaluates to 0. However, the argument is not redundant when we consider arbitrary contexts since, e.g., the call start(zero(1)) does not evaluate to 0, whereas start(zero(0)) does.

Theorem 3 motivated our interest in S-completeness: since it allows us to ensure that locally redundant arguments are redundant, we conclude that S-completeness is also the key property which allows us to formalize a sound transformation technique for eliminating useless arguments based on statically recognizing the local redundancy of those arguments (see Section 5). Now the question of how to single out locally redundant arguments arises. In the following section, we provide some general criteria for ensuring local redundancy of arguments on the basis of the (redundancy of some) positions in the rhs's of program rules, specifically the positions of the rhs's where the arguments of the functions defined in the lhs's 'propagate' to.

4.2 Redundancy of positions

When considering a particular (possibly non-ground) function call, we can observe a weaker notion of redundancy which allows us to consider arbitrary (deeper) positions within the call.

Definition 5 (Redundant position). *Let $\mathcal{R}$ be a TRS and S be a semantics for $\mathcal{R}$. The position $p \in \mathcal{P}os(t)$ is* redundant *w.r.t. $\mathcal{R}$ and S if, for all $s \in \mathcal{T}(\Sigma)$ and $\sigma \in Subst(\Sigma)$, $\mathsf{S}(\sigma(t)) = \mathsf{S}(\sigma(t[s]_p))$.*

We denote by $rpos_{\mathcal{R},\mathsf{S}}(t)$ the set of redundant positions of a term t w.r.t. a TRS $\mathcal{R}$ and a semantics S. The following proposition states that the positions of a term which are below the indices addressing the redundant arguments of any function symbol occurring in t are redundant.

Proposition 3. *Let $\mathcal{R}$ be a TRS, S be a semantics for $\mathcal{R}$, $t \in \mathcal{T}(\Sigma, \mathcal{X})$, $p \in \mathcal{P}os(t)$, $f \in \mathcal{F}$, and $i \in rarg_{\mathcal{R},\mathsf{S}}(f)$. If there exists $q < p$ such that $p = q.i.p'$ and $root(t|_q) = f$, then $p \in rpos_{\mathcal{R},\mathsf{S}}(t)$.*

Definition 6 (Redundant variable). *Let $\mathcal{R}$ be a TRS, S be a semantics for $\mathcal{R}$, and $t \in \mathcal{T}(\Sigma, \mathcal{X})$. The variable $x \in \mathcal{X}$ is redundant in t w.r.t. $\mathcal{R}$ and S if $\mathcal{P}os_x(t) \subseteq rpos_{\mathcal{R},\mathsf{S}}(t)$.*

We say that $p \in \mathcal{P}os(t)$ is a sub-constructor position if for all $q < p$, $root(t|_q) \in \mathcal{C}$. In particular, Λ is a sub-constructor position.

Proposition 4. *Let $\mathcal{R}$ be a TRS,* S *be a semantics for $\mathcal{R}$, $t \in \mathcal{T}(\Sigma, \mathcal{X})$, $p \in \mathcal{P}os(t)$, $f \in \mathcal{F}$, and $i \in lrarg_{\mathcal{R},\mathsf{S}}(f)$. If there exists $q < p$ such that $p = q.i.p'$, $root(t|_q) = f$, and q is a sub-constructor position, then $p \in rpos_{\mathcal{R},\mathsf{S}}(t)$.*

4.3 Using redundant positions for characterizing (local) redundancy

Theorem 1 says that the more restrictive a semantics is, the more redundancies there are for the arguments of function symbols. According to the hierarchy depicted in Figure 1, eval seems to be the most fruitful semantics for analyzing redundant arguments. In the following, we focus on the problem of characterizing the locally redundant arguments w.r.t. eval by studying the redundancy w.r.t. eval of some positions in the rhs's of program rules.

Definition 7 ((f, i)-variable free). *Let $\mathcal{R}$ be a TRS,* S *be a semantics for $\mathcal{R}$, $f \in \Sigma$, $\mathcal{R}_f$ the set of rules for f, and $i \in \{1, \ldots, ar(f)\}$. A subset of rules $\mathcal{S} \subseteq \mathcal{R}_f$ is (f, i)-variable free w.r.t. f and* S *if, for all $l \to r \in \mathcal{S}$ and $x \in \mathcal{V}ar(l|_i)$, x is redundant in r w.r.t. $\mathcal{R}$ and* S*.*

For instance, TRS $\mathcal{R}$ in Example 2 is trivially $(\mathtt{zero}, 1)$-variable free. Now, we are ready to give a first characterization of locally redundant arguments: they correspond to variables which occur at positions which are redundant for evaluating the rhs's of the corresponding program rules.

Theorem 4. *Let $\mathcal{R}$ be a left-linear TRS, $f \in \Sigma$, $\mathcal{R}_f$ be the set of rules for f, and $i \in \{1, \ldots, ar(f)\}$. If $\mathcal{R}_f$ is (f, i)-variable free w.r.t. f and* eval *and, for all $l \to r \in \mathcal{R}_f$, $l|_i \in \mathcal{X}$, then $i \in lrarg_{\mathcal{R},\mathsf{eval}}(f)$.*

Corollary 2. *Let $\mathcal{R}$ be a left-linear TRS, $f \in \Sigma$, $\mathcal{R}_f$ be the set of rules for f, and $i \in \{1, \ldots, ar(f)\}$. If, for all $l \to r \in \mathcal{R}_f$, $l|_i \in \mathcal{X}$ and $l|_i \notin \mathcal{V}ar(r)$, then $i \in lrarg_{\mathcal{R},\mathsf{eval}}(f)$.*

Example 8. Consider the TRS $\mathcal{R}$

$$\mathtt{f(x,y)} \to \mathtt{g(y,s(x))} \qquad \mathtt{g(a,x)} \to \mathtt{a} \qquad \mathtt{g(b,x)} \to \mathtt{b}$$

By Corollary 2, the second argument of $\mathtt{g}$ is locally redundant. According to Proposition 4, variable $\mathtt{x}$ in $\mathtt{g(y,s(x))}$ is redundant. Hence, by Theorem 4, the first argument of $\mathtt{f}$ is locally redundant.

It is interesting to note that Theorem 4 does not hold for a greater semantics like nf.

Example 9. (Continuing Example 8) Since $\mathsf{nf}(\mathtt{f(a,s(a))}) = \{\mathtt{g(s(a),s(a))}\}$ and $\mathsf{nf}(\mathtt{f(b,s(a))}) = \{\mathtt{g(s(a),s(b))}\}$, the first argument of $\mathtt{f}$ is not locally redundant w.r.t. nf.

In Theorem 4, the requirement that $l|_i$ is a variable (in each rule $l \to r \in \mathcal{R}_f$) is quite restrictive. In the following, we provide a different sufficient criterion for local redundancy which is less demanding regarding the shape of the rules, although it requires confluence and eval-completeness.

Definition 8. *Let Σ be a signature, $t = f(t_1, \ldots, t_k)$, $s = f(s_1, \ldots, s_k)$ be linear terms such that $\mathcal{V}ar(t) \cap \mathcal{V}ar(s) = \varnothing$, and $i \in \{1, \ldots, k\}$. We say that t and s* unify up to the i-th argument with *mgu σ if either $k = 1$ (and thus $\sigma = id$) or $k > 1$ and t_j and s_j unify with* mgu *σ_j for $j \in \{1, \ldots, k\} - \{i\}$ (and then $\sigma = \cup_{j \in \{1,\ldots,k\}-\{i\}} \sigma_j$).*

Definition 9 (Redundancy triples). *Let $\mathcal{R}$ be a left-linear TRS, $f \in \mathcal{F}$, and $i \in \{1, \ldots, ar(f)\}$. Given two different (possibly renamed) rules $l \to r$, $l' \to r'$ for f such that $\mathcal{V}ar(l) \cap \mathcal{V}ar(l') = \varnothing$ and $l, l' \in \mathcal{P}att(\Sigma, \mathcal{X})$, we say that $\langle l \to r, l' \to r', \sigma\rangle$ (which we consider equivalent to $\langle l' \to r', l \to r, \sigma\rangle$) is an (f, i)-redundancy triple of $\mathcal{R}$ if l and l' unify up to the i-th argument with* mgu *σ.*

Example 10. Consider the TRS in Example 2. This TRS has a single $(\mathtt{zero}, 1)$-redundancy triple: $\langle \mathtt{zero(0)} \to \mathtt{0}, \mathtt{zero(1)} \to \mathtt{0}, id\rangle$.

Given a left-linear TRS $\mathcal{R}$, a symbol $f \in \Sigma$ and $i \in \{1, \ldots, ar(f)\}$, we denote by $rtriple_{\mathcal{R},i}(f)$ the set of (f, i)-redundancy triples of $\mathcal{R}$. Then, we have the following result: Roughly speaking, if the (ground instances of the) rhs's of the rules in each (f, i)-triple $\langle l \to r, l' \to r', \sigma\rangle$ converge to the same value, then the i-th argument of f is redundant w.r.t. eval.

Theorem 5. *Let $\mathcal{R}$ be a left-linear, confluent, and eval-complete TRS. Let $f \in \mathcal{F}$ and $i \in \{1, \ldots, ar(f)\}$ be such that $\mathcal{R}_f$ is (f, i)-variable free. If for all $\langle l \to r, l' \to r', \sigma\rangle \in rtriple_{\mathcal{R},i}(f)$, and $\varphi \in Subst(\mathcal{C})$, $\mathsf{eval}(\varphi \circ \sigma(r)) = \mathsf{eval}(\varphi \circ \sigma(r'))$, then $i \in rarg_{\mathcal{R},\mathsf{eval}}(f)$.*

Corollary 3. *Let $\mathcal{R}$ be a left-linear, confluent, and eval-complete TRS. Let $f \in \Sigma$, $c \in \Sigma$ be an arbitrary constant, and $i \in \{1, \ldots, ar(f)\}$ be such that $\mathcal{R}$ is (f, i)-variable free. If for all $\langle l \to r, l' \to r', \sigma\rangle \in rtriple_{\mathcal{R},i}(f)$, and*

$$\sigma_c(x) = \begin{cases} \sigma(x) & \text{if } x \notin \mathcal{V}ar(l|_i) \cup \mathcal{V}ar(l'|_i) \\ c & \text{otherwise} \end{cases}$$

terms $\sigma_c(r)$ and $\sigma_c(r')$ have a common reduct, then $i \in rarg_{\mathcal{R},\mathsf{eval}}(f)$.

Example 11. Consider the left-linear, confluent, and eval-complete TRS $\mathcal{R}$:

```
zero(0) → 0            f(0,x) → zero(x)     f(1,0) → 1
zero(1) → 0            f(0,x) → 0           f(1,1) → 1
zero(zero(x)) → 0
```

By Corollary 3, and according to the only triple of $(\mathtt{zero}, 1)$-redundancy showed in Example 10, the first argument of $\mathtt{zero}$ is redundant. By Proposition 3, variable $\mathtt{x}$ in the rhs $\mathtt{zero(x)}$ of the first rule for $\mathtt{f}$ is redundant. Thus, $\mathcal{R}_{\mathtt{f}}$

is $(\mathtt{f},2)$-variable free. Since triple $\langle \mathtt{f(0,x)} \to \mathtt{zero(x)}, \mathtt{f(0,x)} \to \mathtt{0}, id\rangle$ and triple $\langle \mathtt{f(1,0)} \to \mathtt{1}, \mathtt{f(1,1)} \to \mathtt{1}, id\rangle$ are the only $(\mathtt{f},2)$-redundancy triples for `f`, and derivations $\mathtt{zero(0)} \to^* \mathtt{0}$, $\mathtt{0} \to^* \mathtt{0}$, and $\mathtt{1} \to^* \mathtt{1}$ prove that the rhs's of the rules in both triples have a common reduct, by Corollary 3, the second argument of `f` is redundant.

Let us further illustrate the usefulness of Corollary 3 in the detection of the redundant arguments of `applast` from Section 1. The program `applast` is left-linear, confluent, terminating and CD (hence eval-complete). Since $\mathcal{R}_{\mathtt{lastnew}}$ is $(\mathtt{lastnew},1)$-variable free, by Theorem 4 and Theorem 3, the first argument of `lastnew` is redundant w.r.t. eval. Also, it is not difficult to demonstrate that variable `ys` in `lastnew(y,ys,z)` is redundant. Hence, $\mathcal{R}_{\mathtt{lastnew}}$ is $(\mathtt{lastnew},2)$-variable free since, by Proposition 3, variable `y` is redundant in `lastnew(y,ys,z)` and `ys` is proved redundant in the same term. Therefore, since both `lastnew(nil,nil,z)` and `z` rewrite to `z`, by Corollary 3, the second argument of `lastnew` is redundant. Now, since the positions of `x` and `xs` in the rhs of the first rule of `applast` have been proven redundant, $\mathcal{R}_{\mathtt{applast}}$ is $(\mathtt{applast},1)$-variable free and, since `lastnew(nil,nil,z)` and `z` rewrite to `z`, by Corollary 3 we conclude that the first argument of `applast` is redundant.

5 Erasing redundant arguments from TRSs

In this section, we formalize a procedure for removing redundant arguments from a TRS. The basic idea is simple: if an argument of f is redundant, it does not contribute to obtaining the value of any call to f and can be dropped from program R. Hence, we remove redundant formal parameters and corresponding actual parameters for each function symbol in R. We begin with the notion of syntactic erasure which is intended to picking up the redundant arguments of function symbols.

Definition 10 (Syntactic erasure). *A* syntactic erasure *is a mapping $\rho : \Sigma \to \mathcal{P}(\mathbb{N})$ such that for all $f \in \Sigma$, $\rho(f) \subseteq \{1, \ldots, ar(f)\}$. We say that a syntactic erasure ρ is* sound *for a TRS $\mathcal{R}$ and a semantics* S *for $\mathcal{R}$ if, for all $f \in \Sigma$, $\rho(f) \subseteq rarg_{\mathcal{R},\mathsf{S}}(f)$.*

Since we are interested in *removing* redundant arguments from function symbols, we transform the functions by reducing their arity according to the information provided by the redundancy analysis, thus building a new, *erased* signature.

Definition 11 (Erasure of a signature). *Given a signature Σ and a syntactic erasure $\rho : \Sigma \to \mathcal{P}(\mathbb{N})$, the erasure of Σ is the signature Σ_ρ whose symbols $f_\rho \in \Sigma_\rho$ are one to one with symbols $f \in \Sigma$ and whose arities are related by $ar(f_\rho) = ar(f) - |\rho(f)|$.*

Now we extend the procedure to terms in the obvious way.

Definition 12 (Erasure of a term). *Given a syntactic erasure $\rho : \Sigma \to \mathcal{P}(\mathbb{N})$, the function $\tau_\rho : \mathcal{T}(\Sigma, \mathcal{X}) \to \mathcal{T}(\Sigma_\rho, \mathcal{X})$ on terms is: $\tau_\rho(x) = x$ if $x \in \mathcal{X}$ and $\tau_\rho(f(t_1, \ldots, t_n)) = f_\rho(\tau_\rho(t_{i_1}), \ldots, \tau_\rho(t_{i_k}))$ where $\{1, \ldots, n\} - \rho(f) = \{i_1, \ldots, i_k\}$ and $i_m < i_{m+1}$ for $1 \leq m < k$.*

The erasure procedure is extended to TRSs: we erase the lhs's and rhs's of each rule according to τ_ρ. In order to avoid extra variables in rhs's of rules (that arise from the elimination of redundant arguments of symbols in the corresponding lhs), we replace them by an arbitrary constant of Σ (which automatically belongs to Σ_ρ).

Definition 13 (Pre-erasure of a TRS). *Let $\mathcal{R} = (\Sigma, R)$ be a TRS, such that Σ has a constant symbol c, and ρ be a syntactic erasure for Σ. The pre-erasure $\mathcal{R}_\rho$ of $\mathcal{R}$ is $\mathcal{R}_\rho = (\Sigma_\rho, \{\tau_\rho(l) \to \sigma_l(\tau_\rho(r)) \mid l \to r \in \mathcal{R}\})$ where the substitution σ_l for a lhs l is given by $\sigma_l(x) = c$ for all $x \in \mathcal{V}ar(l) - \mathcal{V}ar(\tau_\rho(l))$ and $\sigma_l(y) = y$ whenever $y \in \mathcal{V}ar(\tau_\rho(l))$.*

Example 12. The pre-erasure $\mathcal{R}_\rho$ of the TRS $\mathcal{R}$ in Example 2 consists of a single rule `zero` $\to$ `0` for $\rho($`zero`$) = \{1\}$.

The following definition is only meaningful for terminating term rewriting systems.

Definition 14 (Erasure of a TRS). *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and ρ be a syntactic erasure for Σ. The erasure $\mathcal{R}'_\rho$ of $\mathcal{R}$ is obtained from the pre-erasure $\mathcal{R}_\rho$ of $\mathcal{R}$ by* compression transformation *defined as removing any trivial rule $t \to t$ of $\mathcal{R}_\rho$ and then normalizing the rhs's of the rules.*

Example 13. Consider the program $\mathcal{R}$ for `applast` from Section 1 and the syntactic erasure $\rho($`applast`$) = \{1\}$ and $\rho($`lastnew`$) = \{1, 2\}$. As discussed at the end of Section 4.3, this erasure is sound. The pre-erasure $\mathcal{R}_\rho$ is:

```
applast(z) → lastnew(z)        lastnew(z) → lastnew(z)
applast(z) → z                 lastnew(z) → z
```

and the erasure $\mathcal{R}'_\rho$ is:

```
applast(z) → z                 lastnew(z) → z
```

The following theorems establish the completeness and correctness of the pre-erasure procedure.

Theorem 6 (Completeness). *Let $\mathcal{R}$ be a left-linear TRS, S be a semantics for $\mathcal{R}$, $t \in \mathcal{T}(\Sigma)$, and ρ be a sound syntactic erasure w.r.t. $\mathcal{R}$ and S. If $\delta \in \mathsf{S}(t)$, then $\tau_\rho(t) \to^*_{\mathcal{R}_\rho} \tau_\rho(\delta)$.*

Theorem 7 (Correctness). *Let $\mathcal{R}$ be a left-linear TRS, $t \in \mathcal{T}(\Sigma)$, $\delta \in \mathcal{T}(\mathcal{C})$, and ρ be a sound syntactic erasure w.r.t. $\mathcal{R}$ and eval. If $\tau_\rho(t) \to^*_{\mathcal{R}_\rho} \delta$, then $\delta \in \mathsf{eval}(t)$.*

Corollary 4. *Let $\mathcal{R}$ be a left-linear TRS,* S *be a semantics for* $\mathcal{R}$ *such that* $\mathsf{eval} \preceq \mathsf{S}$, $t \in \mathcal{T}(\Sigma)$, $\delta \in \mathcal{T}(\mathcal{C})$, *and* ρ *be a sound syntactic erasure w.r.t.* $\mathcal{R}$ *and* S. *If* $\tau_\rho(t) \to^*_{\mathcal{R}_\rho} \delta$, *then* $\delta \in \mathsf{S}(t)$.

Since it is well-known that the compression transformation is correct and complete for confluent and terminating programs, we have the following immediate result.

Theorem 8 (Correctness and completeness). *Let* $\mathcal{R}$ *be a left-linear, confluent and terminating TRS,* $t \in \mathcal{T}(\Sigma)$, $\delta \in \mathcal{T}(\mathcal{C})$, *and* ρ *be a sound syntactic erasure w.r.t.* $\mathcal{R}$ *and* eval. *Then,* $\tau_\rho(t) \to^*_{\mathcal{R}'_\rho} \delta$ *if and only if* $\delta \in \mathsf{eval}(t)$.

Finally, we are able to demonstrate that the erasure transformation preserves the (confluence, termination, and CD) behavior of the original TRS (independenly).

Theorem 9. *Let* $\mathcal{R}$ *be a TRS,* S *be a semantics for* $\mathcal{R}$ *and* $\alpha \in \{$*confluence, termination, CD*$\}$. *Let* ρ *be a sound syntactic erasure w.r.t.* $\mathcal{R}$ *and* S. *Then, if* $\mathcal{R}$ *satisfies property* α, *then the property also holds for the erasure of* $\mathcal{R}$, $\mathcal{R}'_\rho$.

References

[AFJV97] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of PEPM'97, Sigplan Notices*, volume 32(12):151–162. ACM Press, New York, 1997.

[AFV98a] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998.

[AFV98b] M. Alpuente, M. Falaschi, and G. Vidal. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 30(3es):9es, 1998.

[AHLV99] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Inductively Sequential Functional Logic Programs. In *Proc. of ICFP'99*, pages 273-283. ACM Press, 1999.

[App92] A.W. Appel. Compiling with Continuations. Cambridge U. Press, 1992.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles Techniques and Tools*. Addison-Wesley, 1986.

[Bel95] F. Bellegarde. ASTRE: Towards a fully automated program transformation system. In *Proc. of RTA'95*, LNCS 914:403–407. Springer-Verlag, Berlin, 1995.

[BN98] F. Baader and T. Nipkow. Term Rewriting and All That. Cambridge University Press, 1998.

[Bon88] A. Bondorf. Towards a Self-Applicable Partial Evaluator for Term Rewriting Systems. In *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 27–50. North-Holland, Amsterdam, 1988.

[CD93] C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of POPL'93*, pages 493–501. ACM, New York, 1993.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, **3**:69-115, 1987.

[DR93] N. Dershowitz and U. Reddy. Deductive and Inductive Synthesis of Equational Programs. *Journal of Symbolic Computation*, 15:467–494, 1993.

[Gal93] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM'93*, pages 88–98. ACM, New York, 1993.

[GS94] R. Glück and M. Sørensen. Partial deduction and driving are equivalent. In *Proc. of PLILP'94*, LNCS 844:165–181. Springer-Verlag, Berlin, 1994.

[Han94] M. Hanus. The integration of functions into logic programming: From theory to practice. *J. Logic Program. 19&20*, 583–628.

[HH80] G. Huet and J.M. Hullot. Proofs by Induction in Equational Theories with Constructors. In *Proc. of FOCS'80*, pages 96-107. IEEE Computer Society, Los Alamitos, California, 1980.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[KKSV95] R. Kennaway, J.W. Klop, M.R. Sleep, and F.-J. de Vries. Transfinite reductions in Orthogonal Term Rewriting Systems. *Information and Computation* 119(1):18-38, 1995.

[Hug88] John Hughes. Backwards Analysis of Functional Programs. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *IFIP Workshop on Partial Evaluation and Mixed Computation*, pages 187–208, 1988.

[Leu98] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Tech. Rep., Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[LG97] L. Lafave and J.P. Gallagher. Constraint-based Partial Evaluation of Rewriting-based Functional Logic Programs. In *Proc. of LOPSTR'97*, LNCS 1463:168–188. Springer-Verlag, Berlin, 1997.

[LS91] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[LS96] M. Leuschel and M. H. Sørensen. Redundant Argument Filtering of Logic Programs. In *Proc of LOPSTR'96*, LNCS 1207:83–103. Springer-Verlag, Berlin, 1996.

[Mid97] A. Middeldorp. Call by Need Computations to Root-Stable Form. In *Proc. of POPL'97*, pages 94-105. ACM Press, 1997.

[MS96] A. Miniussi and D. J. Sherman. Squeezing Intermediate Construction in Equational Programs. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNS 1110:284–302, Springer-Verlag, Berlin,1996.