Removing Redundant Arguments of Functions*

María Alpuente, Santiago Escobar, and Salvador Lucas

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

Abstract. The 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 expression does not change the result. In this work, we provide a method 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 languages. On the basis of the uselessness of redundant arguments, we also propose an erasure procedure which may avoid wasteful computations while still preserving the semantics (under ascertained conditions). A prototype implementation of these methods has been undertaken, which demonstrates the practicality of our approach.

1 Introduction

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 [6, 16, 25, 26]. Examples are redundant arguments in the functions defined by the program, as well as useless program rules.

Example 1. Consider the following program, which calculates the last element of a list and the concatenation of two lists of natural numbers, respectively:

```
data Nat = 0 | S Nat
append::[Nat] -> [Nat] -> [Nat]          last::[Nat] -> Nat
append nil    y = y                      last (x:nil)   = x
append (x:xs) y = x:(append xs y)        last (x:y:ys) = last (y:ys)
```

Assume that we specialize this program for 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, `z`, of the resulting list (the example is borrowed from DPPD library of benchmarks [24] and was also considered in [23, 33] for logic program specialization). Commonly, the optimized program

* Work partially supported by CICYT TIC2001-2705-C03-01, Acciones Integradas HI2000-0161, HA2001-0059, HU2001-0019, and Generalitat Valenciana GV01-424.

which can be obtained by using an automatic specializer of functional programs such as [3–5] is:

```
applast::[Nat] -> Nat -> Nat       lastnew::Nat -> [Nat] -> Nat -> Nat
applast nil    z = z               lastnew x nil    z = z
applast (x:xs) z = lastnew x xs z  lastnew x (y:ys) z = lastnew y ys z
```

This program is too far from {`applast′ ys z = lastnew′ z`, `lastnew′ z = z`}, a more feasible program with the same evaluation semantics, or even the "optimal" program –without redundant parameters– {`applast″ z = z`} which one would ideally expect (here the rule for the "local" function `lastnew′` is disregarded, since it is not useful when the definition of `applast′` is optimized). Indeed, note that the first argument of the function `applast` is redundant (as well as the first and 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 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 [3, 11, 12]. Known procedures for removing dead code such as [7, 19, 26] do not apply to this example either.

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 (TRSs), as a model for the programs that can be written in more sophisticated languages.

At first sight, one could naïvely think that redundant arguments are a straight counterpart of "needed redex" positions, which are well studied in [15]. Unfortunately, this is not true as illustrated by the following example.

Example 2. Consider the optimized program of Example 1 extended with:

```
take:: Nat -> [Nat] -> [Nat]
take 0     xs     = []
take (S n) (x:xs) = x:take n xs
```

The contraction of redex `(take 1 (1:2:[]))` at position 1 in the term t =`applast (take 1 (1:2:[])) 0` (we use 1, 2 instead of `S 0`, `S (S 0)`) is *needed* to normalize t (in the sense of [15]). However, the first argument of `applast` is redundant for normalization, as we showed in Example 1, and the program could be improved by dropping this useless parameter.

In this paper, we provide a semantic characterization of redundancy which is parametric w.r.t. the observed semantics S. After some preliminaries in Section 2, in Section 3 we consider different (reduction) semantics, including the standard normalization semantics (typical of pure rewriting) and the evaluation semantics (closer to functional programming). In Section 4 we introduce the notion of redundancy of an argument w.r.t. a semantics S, and derive a decidability result for the redundancy problem w.r.t. S. Inefficiencies caused by the redundancy of arguments cannot be avoided by using standard rewriting strategies. Therefore, in Section 5 we formalize an elimination procedure which gets rid of the useless arguments and provide sufficient conditions for the preservation of the semantics. Preliminary experiments indicate that our approach is both practical and useful. We conclude with some comparisons with the related literature and future work. Proofs of all technical results are given in [2].

2 Preliminaries

Term rewriting systems provide an adequate computational model for functional languages which allow the definition of functions by means of patterns (e.g., Haskell, Hope or Miranda) [8,17,34]. In the remainder of the paper we follow the standard framework of term rewriting for developing our results (see [8] for missing definitions).

Definitions are given in the one-sorted case. The extension to many-sorted signatures is not difficult [30]. 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 $ar : \Sigma \to \mathbb{N}$. 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 $\bar{t}$. The number k of elements of the tuple $\bar{t}$ will be clarified by the context. Let $\mathit{Subst}(\Sigma,\mathcal{X})$ denote the set of substitutions and $\mathit{Subst}(\Sigma)$ 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 $x \in \mathcal{X}$. By $\mathcal{P}os(t)$ we denote the set of positions of a term 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 $\mathcal{R}$. By $\mathsf{NF}_{\mathcal{R}}$ we denote the set of finite normal forms of $\mathcal{R}$. 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 constructor system (*CS*) is a TRS whose lhs's are patterns. A term t is a head-normal form (or root-stable) if it cannot be rewritten to a redex. The set of head-normal forms of $\mathcal{R}$ is denoted by $\mathsf{HNF}_{\mathcal{R}}$.

3 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. Our notion of semantics is aimed to couch operational as well as denotational aspects.

A *term semantics* for a signature Σ is a mapping $\mathsf{S} : \mathcal{T}(\Sigma) \to \mathcal{P}(\mathcal{T}(\Sigma))$ [28]. A *rewriting semantics* for a TRS $\mathcal{R} = (\Sigma, R)$ is a term semantics S for Σ such that, for all $t \in \mathcal{T}(\Sigma)$ and $s \in \mathsf{S}(t)$, $t \to^*_{\mathcal{R}} s$.

The semantics which is most commonly considered in functional programming is the set of values (ground constructor terms) that $\mathcal{R}$ is able to produce in a finite number of rewriting steps ($\mathsf{eval}_{\mathcal{R}}(t) = \{s \in \mathcal{T}(\mathcal{C}) \mid t \to^*_{\mathcal{R}} s\}$). Other kinds of 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 ($\mathsf{red}_{\mathcal{R}}(t) = \{s \in \mathcal{T}(\Sigma) \mid t \to^*_{\mathcal{R}} s\}$), the set of such reducts that are ground head-normal forms ($\mathsf{hnf}_{\mathcal{R}}(t) = \mathsf{red}_{\mathcal{R}}(t) \cap \mathsf{HNF}_{\mathcal{R}}$), or ground normal forms ($\mathsf{nf}_{\mathcal{R}}(t) = \mathsf{hnf}_{\mathcal{R}}(t) \cap \mathsf{NF}_{\mathcal{R}}$). We also consider the (trivial) semantics empty which assigns an empty set to every term. We often omit $\mathcal{R}$ in the notations for rewriting semantics when it is clear from the context.

The ordering $\preceq$ between semantics [28] provides some interesting properties regarding the redundancy of arguments. Given term semantics S, S' for a signature Σ, we write $\mathsf{S} \preceq \mathsf{S}'$ if there exists $T \subseteq \mathcal{T}(\Sigma)$ such that, for all $t \in \mathcal{T}(\Sigma)$, $\mathsf{S}(t) = \mathsf{S}'(t) \cap T$. We have $\mathsf{empty} \preceq \mathsf{eval}_{\mathcal{R}} \preceq \mathsf{nf}_{\mathcal{R}} \preceq \mathsf{hnf}_{\mathcal{R}} \preceq \mathsf{red}_{\mathcal{R}}$.

Given a rewriting semantics S, it is interesting to determine whether S provides non-trivial information for every input expression. Let $\mathcal{R}$ be a TRS and S be a rewriting semantics for $\mathcal{R}$, we say that $\mathcal{R}$ is S-*defined* if for all $t \in \mathcal{T}(\Sigma)$, $\mathsf{S}(t) \neq \varnothing$ [28]. S-definedness is monotone w.r.t. $\preceq$: if $\mathsf{S} \preceq \mathsf{S}'$ and $\mathcal{R}$ is S-defined, $\mathcal{R}$ is also S'-defined.

S-definedness has already been studied in the literature for different semantics [28]. In concrete, the eval-definedness is related to the standard notion of *completely defined* (CD) TRSs (see [18, 20]). A defined function symbol is completely defined if it does not occur in any ground term in normal form, that is to say functions are reducible on all ground terms (of appropriate sort). A TRS $\mathcal{R}$ is completely defined if each defined symbol of the signature is completely defined. In one-sorted theories, completely defined programs occur only rarely. However, they are common when using types, and each function is defined for all constructors of its argument types.

Let $\mathcal{R}$ be a normalizing (i.e., every term has a normal form) and completely defined TRS; then, $\mathcal{R}$ is $\mathsf{eval}_{\mathcal{R}}$-defined. Being completely defined is sensitive to extra constant symbols in the signature, and so is redundancy; we are not concerned with modularity in this paper.

4 Redundant arguments

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 1 (Redundancy of an argument). *Let* S *be a term semantics for a signature* Σ*,* $f \in \Sigma$*, and* $i \in \{1, \ldots, ar(f)\}$*. The* i*-th argument of* f *is* redundant w.r.t. S *if, for all contexts* $C[\;]$ *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 $rarg_{\mathsf{S}}(f)$ the set of redundant arguments of a symbol $f \in \Sigma$ w.r.t. a semantics S for Σ. Note that every argument of every symbol is redundant w.r.t. empty. Redundancy is antimonotone with regard to the ordering $\preceq$ on semantics.

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

The following result guarantees that constructor symbols have no redundant arguments, which agrees with the common understanding of constructor terms as completely meaningful pieces of information.

Proposition 1. *Let $\mathcal{R}$ be a TRS such that $\mathcal{T}(\mathcal{C}) \neq \varnothing$, and consider S such that $\mathsf{eval}_{\mathcal{R}} \preceq \mathsf{S}$. Then, for all $c \in \mathcal{C}$, $rarg_{\mathsf{S}}(c) = \varnothing$.*

In general, the redundancy of an argument is undecidable. In the following, for a signature Σ, term semantics S for Σ, $f \in \Sigma$, and $i \in \{1, \dots, ar(f)\}$, by "the redundancy problem w.r.t. S", we mean the redundancy of the i-th argument of f w.r.t. S. The following theorem provides a decidability result w.r.t. all the semantics considered in this paper. This result recalls the decidability of other related properties of TRSs, such as the confluence and joinability, and reachability problems (for left-linear, right-ground TRSs) [10, 29].

Theorem 2. *For a left-linear, right-ground TRS $\mathcal{R} = (\Sigma, R)$ over a finite signature Σ, the redundancy w.r.t. semantics $\mathsf{red}_{\mathcal{R}}$, $\mathsf{hnf}_{\mathcal{R}}$, $\mathsf{nf}_{\mathcal{R}}$, and $\mathsf{eval}_{\mathcal{R}}$ is decidable.*

In the following sections, we address the redundancy analysis from a complementary perspective. Rather than going more deeply in the decidability issues, we are interested in ascertaining conditions which (sufficiently) ensure that an argument is redundant in a given TRS. In order to address this problem, we investigate redundancy of positions.

4.1 Redundancy of positions

When considering a particular (possibly non-ground) function call, we can observe a more general notion of redundancy which allows us to consider arbitrary (deeper) positions within the call. We say that two terms $t, s \in \mathcal{T}(\Sigma, \mathcal{X})$ are *p-equal*, with $p \in \mathcal{P}os(t) \cap \mathcal{P}os(s)$ if, for all occurrences w with $w < p$, $t|_w$ and $s|_w$ have the same root.

Definition 2 (Redundant position). *Let S be a term semantics for a signature Σ and $t \in \mathcal{T}(\Sigma, \mathcal{X})$. The position $p \in \mathcal{P}os(t)$ is* redundant *in t w.r.t. S if, for all $t', s \in \mathcal{T}(\Sigma)$ such that t and t' are p-equal, $\mathsf{S}(t') = \mathsf{S}(t'[s]_p)$.*

We denote by $rpos_{\mathsf{S}}(t)$ the set of redundant positions of a term t w.r.t. a semantics S. Note that the previous definition cannot be simplified by getting rid of t' and simply requiring that for all $s \in \mathcal{T}(\Sigma)$, $\mathsf{S}(t) = \mathsf{S}(t[s]_p)$ since redundant positions cannot be analyzed independently if the final goal is to remove the useless arguments, i.e. our notion of redundancy becomes not compositional.

Example 3. Let us consider the TRS $\mathcal{R}$:

$$\mathtt{f(a,a)} \to \mathtt{a} \qquad \mathtt{f(a,b)} \to \mathtt{a} \qquad \mathtt{f(b,a)} \to \mathtt{a} \qquad \mathtt{f(b,b)} \to \mathtt{b}$$

Given the term $\mathtt{f(a,a)}$, for all term $s \in \mathcal{T}(\Sigma)$, $\mathsf{eval}_{\mathcal{R}}(t[s]_1) = \mathsf{eval}_{\mathcal{R}}(t)$ and $\mathsf{eval}_{\mathcal{R}}(t[s]_2) = \mathsf{eval}_{\mathcal{R}}(t)$. However, $\mathsf{eval}_{\mathcal{R}}(t[\mathsf{b}]_1[\mathsf{b}]_2) \neq \mathsf{eval}_{\mathcal{R}}(t)$.

The following result 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 2. *Let* S *be a term semantics for a signature* $\Sigma = \mathcal{F} \uplus \mathcal{C}$, $t \in \mathcal{T}(\Sigma, \mathcal{X})$, $p \in \mathcal{P}os(t)$, $f \in \mathcal{F}$. *For all positions* q, p' *and* $i \in rarg_{\mathsf{S}}(f)$ *such that* $p = q.i.p'$ *and* $root(t|_q) = f$, $p \in rpos_{\mathsf{S}}(t)$ *holds.*

In the following section, we provide some general criteria for ensuring 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 Using redundant positions for characterizing redundancy

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

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

Definition 3 (**(f,i)-redundant variable**)**.** *Let* S *be a term semantics for a signature* Σ, $f \in \mathcal{F}$, $i \in \{1, \dots, ar(f)\}$, *and* $t \in \mathcal{T}(\Sigma, \mathcal{X})$. *The variable* $x \in \mathcal{X}$ *is* (f,i)*-redundant in* t *if* it occurs only at positions $p \in \mathcal{P}os_x(t)$ which are redundant in t, *in symbols* $p \in rpos_{\mathsf{S}}(t)$, *or* it appears in sub-constructor positions of the i-th parameter of f-rooted subterms of t, *in symbols* $\exists q, q'$ *such that* $p = q.i.q'$, $root(t|_q) = f$ *and* q' *is a sub-constructor position of subterm* $t|_{q.i}$.

Note that variables which do not occur in a term t are trivially (f,i)-redundant in t for any $f \in \Sigma$ and $i \in \{1, \dots, ar(f)\}$. Given a TRS $\mathcal{R} = (\Sigma, R)$, we write $\mathcal{R}_f$ to denote the TRS $\mathcal{R}_f = (\Sigma, \{l \to r \in R \mid root(l) = f\})$ which contains the set of rules defining $f \in \mathcal{F}$.

Theorem 3. *Let* $\mathcal{R} = (\mathcal{C} \uplus \mathcal{F}, R)$ *be a left-linear CS. Let* $f \in \mathcal{F}$ *and* $i \in \{1, \dots, ar(f)\}$. *If, for all* $l \to r \in \mathcal{R}_f$, $l|_i$ *is a variable which is* (f,i)*-redundant in* r, *then* $i \in rarg_{\mathsf{eval}_{\mathcal{R}}}(f)$.

Example 4. A standard example in the literature on *useless variable elimination* (UVE) -a popular technique for removing dead variables, see [36, 19]- is the following program[1]:

```
loop(a,bogus,0) → loop(f(a,0),s(bogus),s(0))
loop(a,bogus,s(j)) → a
```

Here it is clear that the second argument does not contribute to the value of the computation. By Theorem 3, the second argument of `loop` is redundant w.r.t. $\mathsf{eval}_{\mathcal{R}}$.

The following example demonstrates that the restriction to constructor systems in Theorem 3 above is necessary.

Example 5. Consider the following TRS $\mathcal{R}$ $\{\mathtt{f(x)}\to\mathtt{g(f(x))},\ \mathtt{g(f(x))}\to\mathtt{x}\}$. Then, the argument 1 of `f(x)` in the lhs of the first rule is a variable which, in the corresponding rhs of the rule, occurs within the argument 1 of a subterm rooted by f, namely `f(x)`. Hence, by Theorem 3 we would have that $1 \in \mathit{rarg}_{\mathsf{eval}_{\mathcal{R}}}(\mathtt{f})$. However, $\mathsf{eval}_{\mathcal{R}}(\mathtt{f(a)}) = \{\mathtt{a}\} \neq \{\mathtt{b}\} = \mathsf{eval}_{\mathcal{R}}(\mathtt{f(b)})$ (considering $\mathtt{a}, \mathtt{b} \in \mathcal{C}$), which contradicts $1 \in \mathit{rarg}_{\mathsf{eval}_{\mathcal{R}}}(\mathtt{f})$.

Using Theorem 3, we are also able to conclude that the first argument of function `lastnew` in Example 1 is redundant w.r.t. $\mathsf{eval}_{\mathcal{R}}$. Unfortunately, Theorem 3 does not suffice to prove that the *second* argument of `lastnew` is redundant w.r.t. $\mathsf{eval}_{\mathcal{R}}$.

In the following, we provide a different sufficient criterion for redundancy which is less demanding regarding the shape of the left hand sides, although it requires orthogonality and eval-definedness, in return. The following definitions are auxiliary.

Definition 4. *Let Σ be a signature, $t = f(t_1, \ldots, t_k)$, $s = f(s_1, \ldots, s_k)$ be terms and $i \in \{1, \ldots, k\}$. We say that t and s* unify up to i-th argument with *mgu σ if $\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_k\rangle$ and $\langle s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k\rangle$ unify mith mgu σ.*

Definition 5 ((f, i)-triple). *Let $\mathcal{R} = (\Sigma, R)$ be a CS, $f \in \Sigma$, and $i \in \{1, \ldots, ar(f)\}$. Given two different (possibly renamed) rules $l \to r$, $l' \to r'$ in $\mathcal{R}_f$ such that $\mathcal{V}ar(l) \cap \mathcal{V}ar(l') = \varnothing$, we say that $\langle l \to r, l' \to r', \sigma\rangle$ is an (f, i)-triple of $\mathcal{R}$ if l and l' unify up to i-th argument with* mgu *σ.*

Example 6. Consider the following CS $\mathcal{R}$ from Example 1:

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

This program has a single (`lastnew`, 2)-triple:
$\langle$`lastnew(x,nil,z)` $\to$ `z`, `lastnew(x,y:ys,z)` $\to$ `lastnew(y,ys,z)`, $id\rangle$

Definition 6 (Joinable (f, i)-triple). *Let $\mathcal{R} = (\mathcal{C} \uplus \mathcal{F}, R)$ be a CS, $f \in \mathcal{F}$, and $i \in \{1, \ldots, ar(f)\}$. An (f, i)-triple $\langle l \to r, l' \to r', \sigma\rangle$ of $\mathcal{R}$ is joinable if $\sigma_{\mathcal{C}}(r)$*

[1] The original example uses natural `100` as stopping criteria for the third argument, while we simplify here to natural `1` in order to code it as `0`/`s` terms.

and $\sigma_{\mathcal{C}}(r')$ are joinable (i.e., they have a common reduct). Here, substitution $\sigma_{\mathcal{C}}$ is given by:

$$\sigma_{\mathcal{C}}(x) = \begin{cases} \sigma(x) \text{ if } x \notin \mathcal{V}ar(l|_i) \cup \mathcal{V}ar(l'|_i) \\ a \ \text{ otherwise, where } a \in \mathcal{C} \text{ is an arbitrary constant of appropriate sort.} \end{cases}$$

Example 7. Consider again the CS $\mathcal{R}$ and the single (`lastnew`, 2)-triple given in Example 6. With ϑ given by $\vartheta = \{$`y` $\mapsto$ `0`, `ys` $\mapsto$ `nil`$\}$, the corresponding rhs's instantiated by ϑ, namely `z` and `lastnew(0,nil,z)`, are joinable (`z` is the common reduct). Hence, the considered (`lastnew`, 2)-triple is joinable.

Roughly speaking, the result below formalizes a method to determine redundancy w.r.t. eval which is based on finding a common reduct of (some particular instances of) the right-hand sides of rules.

Definition 7 ((f,i)-joinable TRS). *Let $\mathcal{R} = (\Sigma, R)$ be a TRS,* S *be a rewriting semantics for $\mathcal{R}$, $f \in \Sigma$, and $i \in \{1,\ldots,ar(f)\}$. $\mathcal{R}$ is (f,i)-joinable if, for all $l \to r \in \mathcal{R}_f$ and $x \in \mathcal{V}ar(l|_i)$, x is (f,i)-redundant in r and all (f,i)-triples of $\mathcal{R}$ are joinable.*

Theorem 4. *Let $\mathcal{R} = (\mathcal{C} \uplus \mathcal{F}, R)$ be an orthogonal and $\mathsf{eval}_{\mathcal{R}}$-defined CS. Let $f \in \mathcal{F}$ and $i \in \{1,\ldots,ar(f)\}$. If $\mathcal{R}$ is (f,i)-joinable then $i \in rarg_{\mathsf{eval}_{\mathcal{R}}}(f)$.*

Joinability is decidable for terminating, confluent TRSs as well as for other classes of TRSs such as right-ground TRSs (see e.g., [29]). Hence, Theorem 4 gives us an effective method to recognize redundancy in CD, left-linear, and (semi-)complete TRSs, as illustrated in the following.

Example 8. Consider again the CS $\mathcal{R}$ of Example 6. This program is orthogonal, terminating and CD (considering sorts), hence is eval-defined. Now, we have the following. The first argument of `lastnew` is redundant w.r.t. $\mathsf{eval}_{\mathcal{R}}$ (Theorem 3). The second argument of `lastnew` is redundant w.r.t. $\mathsf{eval}_{\mathcal{R}}$ (Theorem 4). As a consequence, the positions of variables `x` and `xs` in the rhs of the first rule of `applast` have been proven redundant. Then, since both `lastnew(0,nil,z)` and `z` rewrite to `z`, $\mathcal{R}_{\texttt{applast}}$ is (`applast`, 1)-joinable. By Theorem 4, we conclude that the first argument of `applast` is redundant.

5 Erasing redundant arguments

The presence of redundant arguments within input expressions wastes memory space and can lead to time consuming explorations and transformations (by replacement) of their structure. Redundant arguments are not necessary to determine the result of a function call. At first sight, one could expect that a suitable rewriting strategy which is able to avoid the exploration of redundant arguments of symbols could be defined. In Example 2, we showed that needed reduction is not able to avoid redundant arguments. Context-sensitive rewriting (*csr*) [27], which can be used to forbid reductions on selected arguments of symbols, could also seem adequate for avoiding fruitless reductions at redundant arguments. In *csr*, a replacement map μ indicates the arguments $\mu(f) \subseteq \{1, \ldots, ar(f)\}$ of function symbol f on which reductions are allowed. Let $\mathcal{R}$ be the program `applast` of Example 1 extended with the rules for function `take` of Example 2. If we fix $\mu(\texttt{applast}) = \{2\}$ to (try to) avoid wasteful computations on the first argument of `applast`, using *csr* we are not able to reduce `applast (take 1 (1:2:[])) 0` to `0`.

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 $\mathcal{R}$. Hence, we remove redundant formal parameters and corresponding actual parameters for each function symbol in $\mathcal{R}$. We begin with the notion of syntactic erasure which is intended to pick up redundant arguments of function symbols.

Definition 8 (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 semantics* S *if, for all* $f \in \Sigma$, $\rho(f) \subseteq rarg_{\mathsf{S}}(f)$.

Example 9. Given the signature $\Sigma = \{\texttt{0}, \texttt{nil}, \texttt{s}, \texttt{:}, \texttt{applast}, \texttt{lastnew}\}$ of the TRS $\mathcal{R}$ in Example 6, with $ar(\texttt{0}) = ar(\texttt{nil}) = 0$, $ar(\texttt{s}) = 1$, $ar(\texttt{:}) = ar(\texttt{applast}) = 2$, and $ar(\texttt{lastnew}) = 3$, the following mapping ρ is a *sound syntactic erasure* for the semantics $\mathsf{eval}_{\mathcal{R}}$: $\rho(\texttt{0}) = \rho(\texttt{nil}) = \rho(\texttt{s}) = \rho(\texttt{:}) = \varnothing$, $\rho(\texttt{applast}) = \{1\}$, and $\rho(\texttt{lastnew}) = \{1, 2\}$.

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 9 (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* $ar(f_\rho) = ar(f) - |\rho(f)|$.

Example 10. The erasure of the signature in Example 9 is $\Sigma_\rho = \{\texttt{0}, \texttt{nil}, \texttt{s}, \texttt{:}, \texttt{applast}, \texttt{lastnew}\}$, with $ar(\texttt{0}) = ar(\texttt{nil}) = 0$, $ar(\texttt{s}) = ar(\texttt{applast}) = ar(\texttt{lastnew}) = 1$, and $ar(\texttt{:}) = 2$. Note that, by abuse, we use the same symbols for the functions of the erased signature.

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

Definition 10 (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 11 (Erasure of a TRS). *Let $\mathcal{R} = (\Sigma, R)$ be a TRS, such that Σ has a constant symbol a, and ρ be a syntactic erasure for Σ. The erasure $\mathcal{R}_\rho$ of $\mathcal{R}$ is $\mathcal{R}_\rho = (\Sigma_\rho, \{\tau_\rho(l) \to \sigma_l(\tau_\rho(r)) \mid l \to r \in R\})$ where the substitution σ_l for a lhs l is given by $\sigma_l(x) = a$ 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 11. Let $\mathcal{R}$ be the TRS of Example 6 and ρ be the sound syntactic erasure of Example 9. The erasure $\mathcal{R}_\rho$ of $\mathcal{R}$ consists of the erased signature of Example 10 together with the following rules:

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

Below, we introduce a further improvement aimed to provide the final, "optimal" program.

The following theorem establishes the correctness and completeness of the erasure procedure for the semantics eval.

Theorem 5 (Correctness and Completeness). *Let $\mathcal{R} = (\Sigma, \mathcal{R})$ be a left-linear TRS, ρ be a sound syntactic erasure for* $\mathsf{eval}_\mathcal{R}$*, $t \in \mathcal{T}(\Sigma)$, and $\delta \in \mathcal{T}(\mathcal{C})$. Then, $\tau_\rho(t) \to^*_{\mathcal{R}_\rho} \delta$ iff $\delta \in \mathsf{eval}_\mathcal{R}(t)$.*

In the following we ascertain the conditions for the preservation of some computational properties of TRSs under erasure.

Theorem 6. *Let $\mathcal{R}$ be a left-linear TRS. Let ρ be a sound syntactic erasure for* $\mathsf{eval}_\mathcal{R}$*. If $\mathcal{R}$ is* $\mathsf{eval}_\mathcal{R}$*-defined and confluent, then the erasure $\mathcal{R}_\rho$ of $\mathcal{R}$ is confluent.*

Theorem 7. *Let $\mathcal{R}$ be a left-linear and CD TRS, and ρ be a sound syntactic erasure for* $\mathsf{eval}_\mathcal{R}$*. If $\mathcal{R}$ is normalizing, then the erasure $\mathcal{R}_\rho$ of $\mathcal{R}$ is normalizing.*

In the theorem above, we cannot strengthen normalization to termination. A simple counterexample showing that termination may get lost is the following.

Example 12. Consider the left-linear, (confluent, CD, and) terminating TRS $\mathcal{R}$ {`f(a,y)→a`, `f(c(x),y)→f(x,c(y))`}. The first argument of `f` is redundant w.r.t. $\mathsf{eval}_\mathcal{R}$. However, after erasing the argument, we get the TRS {`f(y) → a`, `f(y) → f(c(y))`}, which is not terminating.

In the example above, note that the resulting TRS is not orthogonal, whereas the original program is. Hence, this example also shows that orthogonality is not preserved under erasure.

The following post-processing transformation can improve the optimization achieved.

Definition 12 (Reduced erasure of a TRS). *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and ρ be a syntactic erasure for Σ. The reduced erasure $\mathcal{R}'_\rho$ of $\mathcal{R}$ is obtained from the erasure $\mathcal{R}_\rho$ of $\mathcal{R}$ by a* compression transformation *defined as removing any trivial rule $t \to t$ of $\mathcal{R}_\rho$ and then normalizing the rhs's of the rules w.r.t. the non-trivial rules of $\mathcal{R}_\rho$.*

Reduced erasures are well-defined whenever $\mathcal{R}_\rho$ is confluent and normalizing since, for such systems, every term has a unique normal form.

Example 13. Let $\mathcal{R}_\rho$ be the erasure of Example 13. The reduced erasure consists of the rules {`applast(z)` $\to$ `z`, `lastnew(z)` $\to$ `z`}.

Since right-normalization preserves confluence, termination and the equational theory (as well as confluence, normalization and the equational theory, in almost orthogonal and normalizing TRSs) [13], and the removal of trivial rules does not change the evaluation semantics of the TRS $\mathcal{R}$ either, we have the following.

Corollary 1. *Let $\mathcal{R}$ be a left-linear TRS, ρ be a sound syntactic erasure for $\mathsf{eval}_\mathcal{R}$, $t \in \mathcal{T}(\Sigma)$, and $\delta \in \mathcal{T}(\mathcal{C})$. If (the TRS which results from removing trivial rules from) $\mathcal{R}_\rho$ is confluent and terminating (alternatively, if it is almost orthogonal and normalizing), then, $\tau_\rho(t) \to^*_{\mathcal{R}'_\rho} \delta$ if and only if $\delta \in \mathsf{eval}_\mathcal{R}(t)$, where $\mathcal{R}'_\rho$ is the reduced erasure of $\mathcal{R}$.*

Erasures and reduced erasures of a TRS preserve left-linearity. For a TRS $\mathcal{R}$ satisfying the conditions in Corollary 1, by using [13], it is immediate that the reduced erasure $\mathcal{R}'_\rho$ is confluent and normalizing. Also, $\mathcal{R}'_\rho$ is CD if $\mathcal{R}$ is.

Hence, let us note that these results allow us to perform the optimization of program `applast` while guaranteeing that the intended semantics is preserved.

6 Conclusion

This paper provides the first results concerning the detection and removal of useless arguments in program functions. We have given a semantic definition of redundancy which takes the semantics S as a parameter, and then we have considered the evaluation semantics (closer to functional programming).

In order to provide practical methods to recognize redundancy, we have ascertained suitable conditions which allow us to simplify the general redundancy problem to the analysis of redundant positions within rhs's of the program rules. These conditions are quite demanding (requiring $\mathcal{R}$ to be orthogonal and $\mathsf{eval}_\mathcal{R}$-defined) but also the optimization which they enable is strong, and powerful. Actually, inefficiencies caused by the redundancy of arguments cannot be avoided by using standard reduction strategies. Therefore, we have developed a transformation for eliminating dead code which appears in the form of useless function calls and we have proven that the transformation preserves the semantics (and the operational properties) of the original program under ascertained conditions. The optimized program that we produce cannot be created as the result of applying standard transformations of functional programming to the original program (such as partial evaluation, supercompilation, and deforestation, see e.g. [32]). We believe that the semantic grounds for redundancy analyses and elimination laid in this paper may foster further insights and developments in the functional programming community and neighbouring fields.

The practicality of our ideas is witnessed by the implementation of a prototype system which delivers encouragingly good results for the techniques deployed in the paper (Sections 4.2 and 5). The prototype has been implemented in PAKCS, the current distribution of the multi-paradigm declarative language Curry [14], and is publicly available at `http://www.dsic.upv.es/users/elp/redargs`.

We have used the prototype to perform some preliminary experiments (available at `http://www.dsic.upv.es/users/elp/redargs/experiments`) which show that our methodology does detect and remove redundant arguments of many common transformation benchmarks, such as `bogus`, `lastappend`, `allzeros`, `doubleflip`, etc (see [24] and references therein). See [2] for details.

6.1 Related Work

Some notions have appeared in the literature of what it means for a term in a TRS $\mathcal{R}$ to be "computationally irrelevant". Our analysis is different from all the related methods in many respects and, in general, incomparable to them.

Contrarily to our notion of redundancy, the meaninglessness of [22, 21] is a property of the terms themselves (they may have meaning in $\mathcal{R}$ or may not), whereas our notion refers to arguments (positions) of function symbols. In [22], Section 7.1, a term t is called *meaningless* if, for each context $C[\,]$ s.t. $C[t]$ has a normal form, we have that $C[t']$ has the same normal form for all terms t'. This can be seen as a kind of superfluity (w.r.t. normal forms) of a fixed expression in any context, whereas our notion of redundancy refers to the possibility of getting rid of some arguments of a given function symbol with regard to some observed semantics. The meaninglessness of [22] is not helpful for the purposes of optimizing programs by removing useless arguments of function symbols which we pursue. On the other hand, terms with a normal form are proven meaningful (i.e., not meaningless) in [22, 21], whereas we might have redundant arguments which are normal forms.

Among the vast literature on analysis (and removal) of unnecessary data structures, the analyses of *unneededness* (or *absence*) of functional programming [9, 16], and the *filtering* of useless arguments and unnecessary variables of logic programming [25, 31] are the closest to our work. In [16], a notion of *needed/unneeded* parameter for list-manipulation programs is introduced which is closely related to the redundancy of ours in that it is capable of identifying whether the value of a subexpression is ignored. The method is formulated in terms of a fixed, finite set of projection functions which introduces some limitations on the class of neededness patterns that can be identified. Since our method gives the information that a parameter is definitely not necessary, our redundancy notion implies Hughes's unneededness, but not vice versa. For instance, constructor symbols cannot have redundant arguments in our framework (Proposition 1), whereas Hughes' notion of unneededness can be applied to the elements of a list: Hughes' analysis is able to determine that in the `length` function (defined as usual), the spine of the argument list is needed but the elements of the list are not needed; this is used to perform some optimizations for the compiler. However, this information cannot be used for the purposes of our work, that is, to remove these elements when the entire list cannot be eliminated.

On the other hand, Hughes's notion of *neededness/unneededness* should not be confused with the standard notion of needed (positions of) redexes of [15]: Example 2 shows that Huet and Levy's neededness does not imply the non-redundancy of the corresponding argument or position (nor vice versa).

The notion of redundancy of an argument in a term rewriting system can be seen as a kind of *comportment property* as defined in [9]. Cousot's comportment analysis generalizes not only the unneededness analyses but also strictness, termination and other standard analyses of functional programming. In [9], comportment is mainly investigated within a denotational framework, whereas our approximation is independent from the semantic formalism.

Proietti and Pettorossi's *elimination procedure* for the removal of unnecessary variables is a powerful unfold/fold-based transformation procedure for logic programs; therefore, it does not compare directly with our method, which would be seen as a post-processing phase for program transformers optimization. Regarding the kind of *unnecessary variables* that the elimination procedure can remove, only variables that occur more than once in the body of the program rule and which do not occur in the head of the rule can be dropped. This is not to say that the transformation is powerless; on the contrary, the effect can be very striking as these kinds of variables often determine multiple traversals of intermediate data structures which are then removed from the program. Our procedure for removing redundant arguments is also related to the Leuschel and Sørensen RAF and FAR algorithms [25], which apply to removing unnecessary arguments in the context of (conjunctive) partial evaluation of logic programs. However, a comparison is not easy either as we have not yet considered the semantics of computed answers for our programs.

People in the functional programming community have also studied the problem of useless variable elimination (UVE). Apparently, they were unaware of the works of the logic programming community, and they started studying the topic from scratch, mainly following a flow-based approach [36] or a type-based approach [7, 19] (see [7] for a discussion of this line of research). All these works address the problem of safe elimination of dead *variables* but heavily handle data structures. A notable exception is [26], where Liu and Stoller discuss how to safely eliminate dead code in the presence of recursive data structures by applying a methodology based on regular tree grammars. Unfortunately, the method in [26] does not apply to achieve the optimization pursued in our running example `applast`.

Obviously, there exist examples (inspired) in the previously discussed works which cannot be directly handled with our results, consider the following TRS:

$$\texttt{length([])} \to \texttt{0} \qquad \texttt{length(x:xs)} \to \texttt{s(length(xs))} \qquad \texttt{f(x)} \to \texttt{length(x:[])}$$

Our methods do not capture the redundancy of the argument of `f`. In [26] it is shown that, in order to evaluate `length(xs)`, we do not need to evaluate the elements of the argument list `xs`. In Liu et al.'s methodology, this means that we could replace the rule for `f` above by `f(_)` $\to$ `length(_:[])` where `_` is a new constant. However, as discussed in Section 5, this could still lead to wasteful computations if, e.g., an eager strategy is used for evaluating the expressions: in that case, a term t within a call `f(`t`)` would be wastefully reduced. Nevertheless, Theorem 3 can be used now with the new rule to recognize the first argument of `f` as redundant. That is, we are allowed to use the following rule: `f` $\to$ `length(_:[])` which completely avoids wasteful computations on redundant arguments. Hence, the different methods are complementary and an enhanced test might be developped by properly combining them.

Recently, the problem of identifying redundant arguments of function symbols has been reduced to proving the validity of a particular class of inductive theorems in the equational theory of confluent, sufficiently complete TRSs. We refer to [1] for details, where a comparison with approximation methods based on abstract interpretation can also be found.

References

1. M. Alpuente, R. Echahed, S. Escobar, S. Lucas. Redundancy of Arguments Reduced to Induction. In *Proc. of WFLP'02*, ENTCS, to appear, 2002.
2. M. Alpuente, S. Escobar, S. Lucas. Removing Redundants Arguments of Functions. Technical report DSIC-II/8/02, UPV, 2002.
3. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of PEPM'97, ACM Sigplan Notices*, volume 32(12):151–162. ACM Press, New York, 1997.
4. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998.
5. M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Inductively Sequential Functional Logic Programs. In Proc. of *ICFP'99, ACM Sigplan Notices*, 34(9):273-283, ACM Press, New York, 1999.
6. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles Techniques and Tools.* Addison-Wesley, 1986.
7. S. Berardi, M. Coppo, F. Damiani and P. Giannini. Type-Based Useless-Code Elimination for Functional Programs. In Walid Taha, editor, *Proc. of SAIG 2000,* LNCS 1924:172–189, Springer-Verlang, 2000.
8. F. Baader and T. Nipkow. Term Rewriting and All That. Cambridge University Press, 1998.
9. P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proc. of ICCL'94*, pages 95–112. IEEE Computer Society Press, Los Alamitos, California, 1994.
10. M. Dauchet, T. Heuillard, P. Lescanne, and S. Tison. Decidability of the Confluence of Finite Ground Term Rewrite Systems and of Other Related Term Rewriting Systems. *Information and Computation*, 88:187-201, 1990.
11. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM'93*, pages 88–98. ACM, New York, 1993.
12. R. Glück and M. Sørensen. Partial deduction and driving are equivalent. In *Proc. of PLILP'94*, LNCS 844:165–181. Springer-Verlag, Berlin, 1994.
13. B. Gramlich. On Interreduction of Semi-Complete Term Rewriting Systems. *Theoretical Computer Science*, 258(1-2):435–451, 2001.
14. M. Hanus. Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~curry`, 2001.
15. G. Huet and J.J. Lévy. Computations in orthogonal term rewriting systems. In J.L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of J. Alan Robinson*, pages 395-414 and 415-443. The MIT Press, Cambridge, MA, 1991.

16. J. 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.
17. J.W. Klop. Term Rewriting Systems. In S. Abramsky, D.M. Gabbay and T.S.E. Maibaum. *Handbook of Logic in Computer Science*, volume 3, pages 1-116. Oxford University Press, 1992.
18. D. Kapur, P. Narendran, and Z. Zhang. On sufficient-completeness and related properties of term rewriting systems. *Acta Informatica* 24:395-416, 1987.
19. N. Kobayashi. Type-based useless variable elimination. In *roc. of PEPM-00*, pages 84-93, ACM Press, 2000.
20. E. Kounalis. Completeness in data type specifications. In B.F. Caviness, editor, *Proc. of EUROCAL'85*, LNCS 204:348-362. Springer-Verlag, Berlin, 1985.
21. R. Kennaway, V. van Oostrom, F.J. de Vries. Meaningless Terms in Rewriting. In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Proc. of ALP'96*, LNCS 1139:254–268. Springer-Verlag, Berlin, 1996.
22. J. Kuper. Partiality in Logic and Computation. Aspects of Undefinedness. PhD Thesis, Universiteit Twente, February 1994.
23. M. Leuschel and B. Martens. Partial Deduction of the Ground Representation and Its Application to Integrity Checking. Tech. Rep. CW 210, K.U. Leuven, 1995.
24. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Tech. Rep., Accessible via `http://www.ecs.soton.ac.uk/~mal/`.
25. 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.
26. Y. A. Liu and S. D. Stoller. Eliminating dead code on recursive data. Science of Computer Programming, 2002. To appear. Preliminary version in *Proc. of SAS'99*, LNCS 1694:211–231. Springer-Verlag, Berlin, 1999.
27. S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1):1-61, January 1998.
28. S. Lucas. Transfinite Rewriting Semantics for Term Rewriting Systems *Proc. of RTA'01*, LNCS 2051:216–230. Springer-Verlag, Berlin, 2001.
29. M. Oyamaguchi. The reachability and joinability problems for right-ground term rewriting systems. *Journal of Information Processing, 13(3)*, pp. 347–354, 1990.
30. P. Padawitz. *Computing in Horn Clause Theories.* EATCS Monographs on Theoretical Computer Science, vol. 16. Springer-Verlag, Berlin, 1988.
31. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *J. Logic Program. 19,20*, 261–320.
32. A. Pettorossi and M. Proietti. A comparative revisitation of some program transformation techniques. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110: 355–385. Springer-Verlag, Berlin, 1996.
33. A. Pettorossi and M. Proietti. A Theory of Logic Program Specialization and Generalization for Dealing with Input Data Properties. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110: 386–408. Springer-Verlag, Berlin, 1996.
34. R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.
35. M. Schütz, M. Schmidt-Schauss and S. E. Panitz. Strictness analysis by abstract reduction using a tableau calculus. In A. Mycroft, editor, *Proc. of SAS'95,* LNCS 983:348–365. Springer-Verlag, 1995.
36. M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *Proc. of POPL'99*, pages 291–302, ACM Press, 1999.