TLP **7** (1&2): 3–35, 2007. © 2007 Cambridge University Press
doi:10.1017/S1471068406002869 Printed in the United Kingdom

Removing redundant arguments automatically

M. ALPUENTE, S. ESCOBAR and S. LUCAS
DSIC, UPV, Camino de Vera s/n, E-46022 Valencia, Spain
(*e-mail:* {alpuente,sescobar,slucas}@dsic.upv.es)

submitted 30 December 2003; revised 12 May 2005; accepted 5 January 2006

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 methods 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.

KEYWORDS: redundant arguments in functions, semantics-preserving program transformation, analysis and optimization, term rewriting

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 (Aho *et al.* 1986; Hughes 1988; Leuschel and Sørensen 1996; Liu and Stoller 2002). Examples are redundant arguments in the functions defined by the program, as well as useless program rules. The notion of redundant argument means that replacing it by whatever expression we like, the final result does not change; independently of actual computations. The following example motivates our ideas.

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

```
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)` ≡ `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 (Leuschel 1998) and was also considered in Leuschel and Martens (1995); Pettorossi and Proietti (1996b) for logic program specialization. Commonly, the optimized program which can be obtained by using an automatic specializer of functional programs (Alpuente *et al.* 1997; Alpuente *et al.* 1998; Alpuente *et al.* 1999) is:

```
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)
```

The first argument of the function applast is redundant (as well as the first and second arguments of the auxiliary function lastnew) and would not typically be written by a programmer who writes this program by hand. This program is far from {applast′(ys,z) = lastnew′(z), lastnew′(z) = z}, a more feasible one 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, after optimizing the definition of applast′, it is not useful anymore). Note that standard (post-specialization) renaming/compression procedures (Alpuente *et al.* 1997; Gallagher 1993; Glück and Sørensen 1994) 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.

Therefore, it seems interesting to formalize program analysis techniques for detecting these kinds of redundancies as well as to formalize transformations for eliminating the dead code that 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 equational, functional, or functional-logic languages. 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 S, including the standard normalization semantics (typical of pure rewriting) and the evaluation semantics (closer to functional and equational programming). In Section 4 we introduce the notion of redundancy of an argument w.r.t. a semantics S and provide some useful properties. In Section 5 we derive a decidability result for the redundancy problem w.r.t. S and provide the first effective method for detecting redundancies, which is based on approximation techniques. Then, in Section 6 we provide a more practical method to recognize redundancy which allows us to simplify the general redundancy problem to the analysis of the rhs's of the program rules.

At first sight, one could naïvely think that redundant arguments are a straight counterpart of "needed redex" positions (Huet and Lévy 1991), a well-known operational notion in term rewriting, which could be easily neutralized by appropriately driving the computation. Unfortunately, this is not true as illustrated by the following example.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

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

```
take(0,xs)      = nil
take(s(n),x:xs) = x:take(n,xs)
```

The contraction of redex `take(1,1:2:nil)` at position 1 in the term[1] t = `applast(take(1,1:2:nil),0)` is *needed* to normalize the term t to the constructor normal form `0`. This means that such redex position (or one of its residuals) must be reduced in each rewriting sequence from t to its normal form `0` (see Huet and Lévy (1991)). 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. Therefore, although needed redexes are an essential piece of the computational process which implements the evaluation, from a semantic point of view, they can be irrelevant (*redundant*).

Since needed redexes must *all* be reduced in *any* reduction sequence leading to a normal form, Example 2 shows that no normalizing reduction strategy is able to dodge the problem by *avoiding* the exploration of the redundant argument. Thus, in general, inefficiencies caused by the redundancy of arguments cannot be avoided by using rewriting strategies. Therefore, in Section 7 we formalize an *elimination* procedure which gets rid of the redundant arguments and provide sufficient conditions for the preservation of the semantics. Preliminary experiments in Section 8 indicate that our approach is both practical and useful.

An extensive comparison with the related literature is provided in Section 9. We summarize some relevant ideas as follows. Strictness analysis[2] (Burn *et al.* 1986; Burn 1991; Jensen 1991; Mycroft 1980; Mycroft and Norman 1992; Sekar *et al.* 1990; Wadler and Hughes 1987) can be used to determine whether the evaluation of an argument e_i within an expression $e = f(e_1, \ldots, e_i, \ldots, e_k)$ is "strictly" necessary to obtain the value of e. The counterpart of this notion has been studied in a number of different analysis techniques such as *dead code analysis* (Liu and Stoller 2002), *unneededness analysis* (Hughes 1988), *absence analysis* (Cousot and Cousot 1994), *filtering analysis* (Leuschel and Sørensen 1996), or *useless analysis* (Wand and Siveroni 1999). Also, similar techniques to detect and remove parts of a program which are computationally irrelevant have been investigated in the past: program specialization (Alpuente *et al.* 1997; Alpuente *et al.* 1998; Alpuente *et al.* 1999; Leuschel and Martens 1995; Pettorossi and Proietti 1994; Pettorossi and Proietti 1996a), slicing (Gouranton 1998; Schoenig and Ducasse 1996; Reps and Turnidge 1996; Szilagyi et al. 2002; Tip 1995; Weiser 1984), compile-time garbage collection (Jones and Métayer 1989; Park and Goldberg 1992; Knoop *et al.* 1994), and dead code removal (Berardi *et al.* 2000; Kobayashi 2000; Liu and Stoller 2002).

[1] In this paper, naturals `1`, `2`,... are often used as shorthand to numbers $\mathtt{s}^n\mathtt{(0)}$ where $n = 1, 2, \ldots$.

[2] Roughly speaking, a function symbol f is strict in its i-th argument if any subterm at such argument position must be completely evaluated during the evaluation of f. In symbols: let $D_1, \ldots, D_k, D$ be ordered sets with least elements $\bot_1, \ldots, \bot_k, \bot$ respectively, expressing undefinedness, a mapping $f : D_1 \times \cdots \times D_k \to D$ is said to be *strict* in its i-th argument if $f(d_1, \ldots, \bot_i, \ldots, d_k) = \bot$ for all $d_1 \in D_1, \ldots, d_k \in D_k$.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

In Section 10, we briefly discuss the detection of redundant arguments in functional logic programs mechanized by narrowing. We conclude in Section 11. Proofs of all technical results are given in (Alpuente *et al.* 2006). This paper is a revised and improved version of (Alpuente *et al.* 2002b).

2 Preliminaries

Term rewriting systems provide an adequate computational model for functional and equational programming languages which allow the definition of functions by means of patterns, e.g., Haskell, Hope, or Miranda (Baader and Nipkow 1998; Klop 1992; Plasmeijer and van Eekelen 1993). In the rest of the paper we follow the standard framework of term rewriting for developing our results; see Baader and Nipkow (1998) and TeReSe (2003) for missing definitions. To simplify our presentation, definitions are given in the one-sorted case; the extension to many-sorted signatures is not difficult (Padawitz 1988), and we comment where they matter the non-obvious details.

Let $\rightarrow \subseteq A \times A$ be a binary relation on a set A. We denote the inverse of $\rightarrow$ by $\leftarrow$, the symmetric closure by $\leftrightarrow$, the transitive closure by $\rightarrow^+$, the reflexive and transitive closure by $\rightarrow^*$, and the reflexive, symmetric and transitive closure by $\leftrightarrow^*$. We say that $\rightarrow$ is *confluent* if, for every $a, b, c \in A$, whenever $a \rightarrow^* b$ and $a \rightarrow^* c$, there exists $d \in A$ such that $b \rightarrow^* d$ and $c \rightarrow^* d$. We say that $\rightarrow$ is *terminating* (or *well-founded*) iff there is no infinite sequence $a_1 \rightarrow a_2 \rightarrow a_3 \cdots$.

Throughout the paper, $\mathcal{X}$ denotes a countable set of variables $\{\mathtt{x}, \mathtt{y}, \mathtt{w}, \ldots\}$, and $\mathcal{F}$ denotes a finite set of function symbols $\{\mathtt{f}, \mathtt{g}, \mathtt{h}, \ldots\}$, each one having a fixed arity given by a function $ar : \mathcal{F} \rightarrow \mathbb{N}$. By $\mathcal{T}(\mathcal{F}, \mathcal{X})$ we denote the set of terms and by $\mathcal{T}(\mathcal{F})$ the set of ground terms, i.e., terms without variable occurrences. $\mathcal{V}ar(t)$ is the set of variables in t. 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.

A *substitution* is a mapping $\sigma : \mathcal{X} \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{X})$ which homomorphically extends to a mapping $\sigma : \mathcal{T}(\mathcal{F}, \mathcal{X}) \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{X})$. The substitution σ is usually different from the identity, i.e., $\forall x \in \mathcal{X} : id(x) = x$, for a finite subset $\mathcal{D}om(\sigma) \subseteq \mathcal{X}$, called the *domain* of σ. By $\theta \circ \sigma$ we denote the composition of the substitutions σ and θ, i.e., $\theta \circ \sigma(x) = \theta(\sigma(x))$. Let $Subst(\mathcal{F}, \mathcal{X})$ denote the set of substitutions and $Subst(\mathcal{F})$ be the set of ground substitutions, i.e., substitutions on $\mathcal{T}(\mathcal{F})$. If $\sigma(t)$ is a ground term, we call σ a *grounding substitution* for t. A *unifier* of two terms t, s is a substitution σ such that $\sigma(t) = \sigma(s)$ and σ is idempotent, i.e., $\sigma \circ \sigma = \sigma$. 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 $\sigma' = \theta \circ \sigma$. By $\sigma|_V$ we denote the restriction of subsitution σ to the variables in V.

Terms are viewed as labelled trees in the usual way. Positions $p, q, \ldots$ are defined as sequences of positive natural numbers used to address subterms of t, with Λ the root position (i.e., the empty sequence), $p.q$ the position concatenation, and $p < q$ the usual prefix ordering. Two positions p, q are disjoint, denoted by $p \parallel q$, if neither $p < q$, $p > q$, nor $p = q$. The symbol labeling the root position of t is denoted as $root(t)$. The subterm at position p of t is denoted as $t|_p$ and $t[s]_p$ is the term t

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

with the subterm at position p replaced by s. The restriction of a set of positions P w.r.t. a position p is defined as $P|_p = \{p' \mid \exists q \in P \wedge q = p.p'\}$, the concatenation of a position p and a set of positions P is defined as $p.P = \{p.q \mid q \in P\}$, and the comparison of a set of positions P w.r.t. a position p is defined as $p \leqslant P$ iff $p \leqslant q$ for each $q \in P$. By $\mathscr{P}os_S(t)$ we denote all positions in t with a symbol or variable from $S \subseteq \mathscr{F} \cup \mathscr{X}$. We use $\mathscr{P}os_f(t)$ and $\mathscr{P}os(t)$ as shorthands for $\mathscr{P}os_{\{f\}}(t)$ and $\mathscr{P}os_{\mathscr{F}\cup\mathscr{X}}(t)$, respectively. A context is a term C with zero or more 'holes', i.e., the fresh constant symbol $\Box$. We usually write simply $C[\ ]$ to denote an arbitrary context, clarifying the number and location of holes 'in situ'. 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[3] $l \to r$, with $l, r \in \mathscr{T}(\mathscr{F}, \mathscr{X})$, $l \notin \mathscr{X}$ and $\mathscr{V}ar(r) \subseteq \mathscr{V}ar(l)$. The left-hand side (*lhs*) of the rule is l and r is the right-hand side (*rhs*). A TRS is a pair $\mathscr{R} = (\mathscr{F}, R)$ where R is a set of rewrite rules and $\mathscr{F}$ is called the signature. A term t rewrites to s (at position p), written $t \to_{\mathscr{R}} s$ (or just $t \to s$), if $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$, for some rule $l \to r \in R$, $p \in \mathscr{P}os(t)$ and substitution σ. An instance $\sigma(l)$ of the *lhs* of a rule $l \to r$ is called a redex; similarly subterm $t|_p$ in a rewrite step is also called a redex. A term t without redexes is said a *normal form*. By $\mathsf{NF}_{\mathscr{R}}$ we denote the set of finite normal forms w.r.t. $\mathscr{R}$. A term t is said a *head-normal form* (or *root-stable*) if it cannot be rewritten to a redex. By $\mathsf{HNF}_{\mathscr{R}}$ we denote the set of head-normal forms w.r.t. $\mathscr{R}$.

A TRS $\mathscr{R}$ is *left linear* if all its lhs's are linear terms. A TRS $\mathscr{R}$ is *ground* (resp. *right-ground*) if all its lhs's and rhs's (resp. only its rhs's) are ground terms. A TRS $\mathscr{R}$ is *terminating* (resp. *confluent*) if the relation $\to_{\mathscr{R}}$ is terminating (resp. confluent). Two terms t, s are *joinable*, denoted by $t \downarrow s$, if there exists a term u such that $t \to^* u$ and $s \to^* u$.

Given $\mathscr{R} = (\mathscr{F}, R)$, we assume $\mathscr{F}$ can be always considered as the disjoint union $\mathscr{F} = \mathscr{C} \uplus \mathscr{D}$ of symbols $c \in \mathscr{C}$, called *constructors*, and symbols $f \in \mathscr{D}$, called *defined functions*, where $\mathscr{D} = \{f \mid f(\bar{l}) \to r \in R\}$ and $\mathscr{C} = \mathscr{F} - \mathscr{D}$. Then, $\mathscr{T}(\mathscr{C}, \mathscr{X})$ is the set of constructor terms. A pattern is a term $f(l_1, \ldots, l_n)$ such that $f \in \mathscr{D}$ and $l_1, \ldots, l_n \in \mathscr{T}(\mathscr{C}, \mathscr{X})$. A constructor system (*CS*) is a TRS whose lhs's are patterns.

Two (possibly renamed) rules $l \to r$ and $l' \to r'$ *overlap*, if there is a non-variable position $p \in \mathscr{P}os_{\mathscr{F}}(l)$ and a most-general unifier σ such that $\sigma(l|_p) = \sigma(l')$. The pair $\langle \sigma(l)[\sigma(r')]_p, \sigma(r) \rangle$ is called a *critical pair* and is also called an *overlay* if $p = \Lambda$. A critical pair $\langle t, s \rangle$ is *trivial* if $t = s$. A left-linear TRS without critical pairs is called *orthogonal*. Note that orthogonality of a TRS $\mathscr{R}$ implies confluence of $\to_{\mathscr{R}}$. A left-linear TRS where its critical pairs are trivial overlays is called *almost orthogonal*.

3 Semantics

The redundancy of an argument of a function f in a TRS $\mathscr{R}$ depends on the semantics properties of $\mathscr{R}$ that we are interested in observing. Our notion of semantics is aimed to couch operational as well as denotational aspects.

[3] We will use also $l = r$ to differentiate a rule from a rewriting step.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

A *term semantics* for a signature $\mathcal{F}$ is a mapping $\mathsf{S} : \mathcal{T}(\mathcal{F}) \to \mathcal{P}(\mathcal{T}(\mathcal{F}))$ (Lucas 2001) which associates a set of terms to a term. A *rewriting semantics* for a TRS $\mathcal{R} = (\mathcal{F}, R)$ is a term semantics S for $\mathcal{F}$ such that, for all $t \in \mathcal{T}(\mathcal{F})$ and $s \in \mathsf{S}(t)$, $t \to^*_{\mathcal{R}} s$, i.e., a term semantics where the set of terms associated to a term is determined only by the program.

The rewriting 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 rewriting 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 ($\mathsf{red}_{\mathcal{R}}(t) = \{s \in \mathcal{T}(\mathcal{F}) \mid t \to^*_{\mathcal{R}} s\}$), 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. Furthermore, a rewriting semantics S for a TRS $\mathcal{R}$ is called ($\mathcal{R}$-)*normalized* if, for all $t \in \mathcal{T}(\mathcal{F})$, $\mathsf{S}(t) \subseteq \mathsf{NF}_{\mathcal{R}}$, i.e., the semantics associates only normal forms to a term. eval and nf are examples of normalized semantics whereas hnf and red are not normalized.

The ordering $\preceq$ between semantics (Lucas 2001) provides some interesting properties regarding the redundancy of arguments. Given term semantics S and S' for a signature $\mathcal{F}$, we write $\mathsf{S} \preceq \mathsf{S}'$ if there exists $T \subseteq \mathcal{T}(\mathcal{F})$ (called *window set* of S' w.r.t. S) such that, for all $t \in \mathcal{T}(\mathcal{F})$, $\mathsf{S}(t) = \mathsf{S}'(t) \cap T$. Note that, then, 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 $t \in \mathcal{T}(\mathcal{F})$, $\mathsf{S}(t) \neq \emptyset$ (Lucas 2001). 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 (Lucas 2001). In concrete, nf-defined TRSs are known as normalizing TRSs (i.e., every term has a normal form (Baader and Nipkow 1998)) and eval-definedness is related to termination and the standard notion of *completely defined* (CD) TRSs; see Kapur *et al.* (1987) and Kounalis (1985). Roughly speaking, a defined function symbol is completely defined if it does not occur in any ground term in normal form, that is to say that 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 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. Thus, we are not concerned with modularity in this work.

From now on, we formulate the notion of a redundant argument and provide some useful properties and detection techniques.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

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 $\mathscr{F}$, $f \in \mathscr{F}$, 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 $t,s \in \mathscr{T}(\mathscr{F})$ 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 $f \in \mathscr{F}$ w.r.t. a semantics S for $\mathscr{F}$. Note that every argument of every symbol is redundant w.r.t. empty. The following result shows that redundancy is antimonotone with regard to the ordering $\preceq$ on semantics.

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

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

Proposition 1 (*Non-redundancy of constructors*)
Let $\mathscr{R}$ be a TRS such that $|\mathscr{T}(\mathscr{C})| > 1$, and consider a rewriting semantics S such that $\mathsf{eval}_{\mathscr{R}} \preceq \mathsf{S}$. Then, for all $c \in \mathscr{C}$, $rarg_{\mathsf{S}}(c) = \emptyset$.

For many-sorted signatures, we would require that $|\mathscr{T}(\mathscr{C})_\tau| > 1$ for the sort τ of an argument of a constructor symbol c. In the following section, we consider several aspects about decidability of the redundancy of an argument.

5 Decidability Issues

In general, the redundancy of an argument is undecidable. However, we are able to provide a decidability result about redundancy w.r.t. all the non-trivial semantics considered in this paper. In this section, for a signature $\mathscr{F}$, term semantics S for $\mathscr{F}$, $f \in \mathscr{F}$, and $i \in \{1,\ldots,ar(f)\}$, by "redundancy w.r.t. S" we mean the redundancy of the i-th argument of f w.r.t. S.

We follow the "(W)SkS approach" to decide a given property P, which is based on ascertaining the conditions for expressing P in a *decidable* logic, namely *the (weak) second-order monadic logic with k successors (W)SkS*; see Thomas (1990). The following theorem by Rabin is the key element for our results in this section.

Theorem 2 (*Robin 1969*)
The (weak) monadic second-order theory of k successor functions *(W)SkS* is decidable.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

First, we recall some basic definitions about the WSkS logic; see e.g., Thomas (1990). *Terms* of the WSkS logic are formed out of individual variables $x, y, z, \ldots$, the empty string Λ, and right concatenation with $1, \ldots, k$. Atomic formulas are equations between terms, inequations $w < w'$ between terms, or expressions $w \in X$ where w is a term and X is a (second-order) variable. Formulas are built from atomic formulas using the logical connectives $\wedge, \vee, \Rightarrow, \neg, \ldots$ and the quantifiers $\exists, \forall$ of both individual and second-order variables. Individual variables are interpreted as elements of $\{1, \ldots, k\}^*$ and second-order variables as finite subsets of $\{1, \ldots, k\}^*$. Equality is the string equality and inequality is the strict prefix ordering. Finite union and intersection, as well as inclusion and equality of sets, are definable in WSkS in an obvious way.

Let us relate TRSs and WSkS logic. Given a finite signature $\mathcal{F}$, let k be the maximal arity of all the function symbols in $\mathcal{F}$ and n be the cardinality of $\mathcal{F}$. A term t is represented in WSkS using $n+1$ set variables X and $X_f, f \in \mathcal{F}$, which are denoted by $\vec{X}$ in the following. X will be the set of all positions of t, and X_f will be the set of positions that are labeled with the corresponding function symbol. The following WSkS formula expresses that $\vec{X}$ encodes a term in $\mathcal{T}(\mathcal{F})$ (Comon 2000; Durand and Middeldorp 1997):

$$\begin{aligned} Term_{\mathcal{F}}(\vec{X}) \stackrel{def}{=} & X = \textstyle\bigcup_{i=1}^{n} X_{f_i} \wedge \bigwedge_{i \neq j} (X_{f_i} \cap X_{f_j} = \emptyset) \\ & \wedge \forall x \in X \; \forall y < x \, (y \in X) \\ & \wedge \; \bigwedge_{f \in \mathcal{F}} (\forall x \in X_f : \textstyle\bigwedge_{l=1}^{ar(f)} (x.l \in X) \wedge \bigwedge_{l=ar(f)+1}^{k} (x.l \notin X)) \end{aligned}$$

If $Term_{\mathcal{F}}(\vec{T})$ holds, then we let $t_{\vec{T}}$ define the term in $\mathcal{T}(\mathcal{F})$ which is uniquely determined by $\mathcal{P}os(t) = T$ and $root(t|_p) = f$ if $p \in T_f$ for all $p \in T$. A subset of ground terms $L \subseteq \mathcal{T}(\mathcal{F})$ is called WSkS *definable* if there exists a WSkS formula Φ with free variables $\vec{T}$ such that $L = \{t_{\vec{T}} \mid Term_{\mathcal{F}}(\vec{T}) \wedge \Phi(\vec{T})\}$.

An arbitrary term semantics S can be encoded as a relation $\mathcal{S}$ between terms: $\mathcal{S} = \{(t, s) \mid t \in \mathcal{T}(\mathcal{F}) \wedge s \in \mathsf{S}(t)\}$. Hence, we say that semantics S is WSkS definable if there exists a WSkS formula Φ with free variables $\vec{T}$ and $\vec{S}$ such that $(t_{\vec{T}}, s_{\vec{S}}) \in \mathcal{S} \Leftrightarrow Term_{\mathcal{F}}(\vec{T}) \wedge Term_{\mathcal{F}}(\vec{S}) \wedge \Phi(\vec{T}, \vec{S})$.

Theorem 3 (*Decidability of redundancy*)
Let S be a term semantics for a signature $\mathcal{F}$. If S is WSkS definable, then redundancy w.r.t. S is decidable.

The following result shows that decidability of redundancy is *antimonotone* with regard to the ordering $\preceq$ on semantics.

Proposition 2
Let S, S' be term semantics for a signature $\mathcal{F}$. If $\mathsf{S} \preceq \mathsf{S}'$, S' is WSkS definable, and there exists a window set $T \subseteq \mathcal{T}(\mathcal{F})$ of S' w.r.t. S which is WSkS definable, then S is WSkS definable.

In Dauchet *et al.* (1990) and Dauchet *et al.* (1987), ground (finite) tree transducers (GTT for short) were introduced to recognize the rewrite relation $\rightarrow_{\mathcal{R}}^*$ in (left-linear and right-)ground TRSs. Since GTT-recognizable relations are definable in WSkS (Comon 2000), the semantics red is also WSkS definable, hence the redundancy

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

w.r.t. red is decidable. Now, the following result shows that the window set $\mathsf{HNF}_{\mathscr{R}}$ is WS*k*S definable; this is useful for proving that semantics hnf is also WS*k*S definable.

Theorem 4
The set $\mathsf{HNF}_{\mathscr{R}}$ of a finite left-linear, right-ground TRS $\mathscr{R}$ is WS*k*S definable.

Then, the following theorem provides the first decidability result w.r.t. all the non-trivial semantics considered in this paper.

Theorem 5 (*Decidability for semantics* $\mathsf{red}_{\mathscr{R}}$, $\mathsf{hnf}_{\mathscr{R}}$, $\mathsf{nf}_{\mathscr{R}}$, *and* $\mathsf{eval}_{\mathscr{R}}$)
For a left-linear, right-ground TRS $\mathscr{R}$ over a finite signature $\mathscr{F}$, the redundancy w.r.t. semantics $\mathsf{red}_{\mathscr{R}}$, $\mathsf{hnf}_{\mathscr{R}}$, $\mathsf{nf}_{\mathscr{R}}$, and $\mathsf{eval}_{\mathscr{R}}$ is decidable.

This result recalls the decidability of other related properties of TRSs, such as confluence, joinability, and reachability problems (for left-linear, right-ground TRSs) (Dauchet *et al.* 1987; Oyamaguchi 1990). For instance, the confluence problem was shown to be undecidable for right-ground TRSs, while it is decidable for ground TRSs and also for left-linear and right-ground TRSs (Dauchet *et al.* 1987). Note that we cannot weaken in our approach the requirement of right-groundness in Theorem 5 to the more general conditions of shallowness (Comon 2000) or growingness (Jacquemard 1996) as the induced rewrite relations are not expressible in the logic WS*k*S that we use to decide the property (Durand and Middeldorp 1997).

In the following section we provide the first redundancy detection method, which (sufficiently) ensures that an argument is redundant in a given TRS.

5.1 Approximations of redundancy

Whenever a property is undecidable or costly to decide, we use approximations. A notion of approximation (for TRSs) that has been proven useful for approximating interesting properties in term rewriting (namely neededness of redexes for normalization) is the following Durand and Middeldorp (1997) and Jacquemard (1996): Given TRSs $\mathscr{R}$ and $\mathscr{R}'$ (possibly with extra variables) over the same signature, $\mathscr{R}'$ *approximates* $\mathscr{R}$ if $\to^*_{\mathscr{R}} \subseteq \to^*_{\mathscr{R}'}$ and $\mathsf{NF}_{\mathscr{R}} = \mathsf{NF}_{\mathscr{R}'}$. An approximation of TRSs is a mapping α from TRSs to TRSs with the property that TRS $\alpha(\mathscr{R})$ approximates TRS $\mathscr{R}$ (Durand and Middeldorp 1997). We write $\mathscr{R}_\alpha$ instead of $\alpha(\mathscr{R})$ to denote the approximation of $\mathscr{R}$ according to α. *Strong*, *nv* (Durand and Middeldorp 1997), *shallow* (Comon 2000), and *growing* (Jacquemard 1996) are examples of such approximations of TRSs. In all these approximations, the rhs's of the rules are modified in different ways. For instance, given a TRS $\mathscr{R}$, $\mathscr{R}_{nv}$ is obtained by replacing *all* variables in the rhs by new, different variables that do *not* occur in the lhs; this is possible since the framework deals with extra variables.

To approximate redundancy, we need to use a new symbol Ω to represent all ground terms (in particular, to be used at the argument position which is tested for redundancy). Inspired by (Durand and Middeldorp 1997; Oyamaguchi 1986), we define our notion of approximation as follows. Let $\mathscr{R}$ be a TRS over a signature $\mathscr{F}$ and $\mathscr{R}'$ be a TRS over the signature $\mathscr{F} \cup \{\Omega\}$, where Ω is a new constant symbol defined by the rules $\{\Omega \to f(\overline{\Omega}) \mid f \in \mathscr{F}\}$. We extend the approximation notion

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

of Durand and Middeldorp (1997) and Jacquemard (1996) naturally to TRSs over signatures $\mathcal{F}$ and $\mathcal{F} \cup \{\Omega\}$, where Ω is a special symbol that potentially expresses any term. Note that we consider the normalization semantics only for ground terms. Thus, we say that $\mathcal{R}'$ approximates $\mathcal{R}$ (but notice that, now, $\mathcal{R}'$ is a TRS on $\mathcal{F} \cup \{\Omega\}$) if $\to^*_{\mathcal{R}} \cap (\mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F})) \subseteq \to^*_{\mathcal{R}'} \cap (\mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F}))$ and $\mathsf{NF}_{\mathcal{R}} = \mathsf{NF}_{\mathcal{R}'}$. Note that, $\to^*_{\mathcal{R}} \subseteq (\mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{T}(\mathcal{F}, \mathcal{X}))$ whereas $\to^*_{\mathcal{R}'} \subseteq (\mathcal{T}(\mathcal{F} \cup \{\Omega\}, \mathcal{X}) \times \mathcal{T}(\mathcal{F} \cup \{\Omega\}, \mathcal{X}))$; however, by definition of $\mathcal{R}'$, $\mathsf{NF}_{\mathcal{R}'} \subseteq \mathcal{T}(\mathcal{F})$.

The following notation is auxiliary.

Definition 2 (S-*determinacy w.r.t.* f *and* i)
Given a symbol $f \in \mathcal{F}$ and an argument $i \in \{1, \ldots, ar(f)\}$, we say that the semantics S is *determined* w.r.t. f and i if for every context $C[\,]$ and $t \in \mathcal{T}(\mathcal{F})$ such that $root(t) = f$, then $|\mathsf{S}(C[t[\Omega]_i])| \leqslant 1$; where $|A|$ stands for the cardinality of the set A.

The following theorem provides a sufficient condition for redundancy which is the basis of our decidable approximations of redundancy.

Theorem 6 (*Approximation of redundancy*)
Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS, $\mathcal{R}'$ be an approximation of $\mathcal{R}$, $f \in \mathcal{F}$, $i \in \{1, \ldots, ar(f)\}$, and $\mathsf{S} \in \{\mathsf{eval}, \mathsf{nf}\}$. If $\mathcal{R}$ is $\mathsf{S}_{\mathcal{R}}$-defined and $\mathsf{S}_{\mathcal{R}'}$ is determined w.r.t. f and i, then $i \in rarg_{\mathsf{S}_{\mathcal{R}}}(f)$.

It is an open problem whether redundancy is decidable for terminating TRSs. Nevertheless, Theorem 6 ensures that redundancy w.r.t. nf is approximable for terminating TRSs, since any terminating TRS $\mathcal{R}$ is $\mathsf{nf}_{\mathcal{R}}$-defined. The following theorem ensures that WSkS definability of a semantics entails the possibility of guaranteeing decidability of a given approximation.

Theorem 7 (*Decidability of* S-*determinacy w.r.t.* f *and* i)
Let S be a term semantics for a signature $\mathcal{F} \cup \{\Omega\}$. If S is WSkS definable, then it is decidable whether S is determined w.r.t. f and i.

Remember that the semantics $\mathsf{eval}_{\mathcal{R}}$ and $\mathsf{nf}_{\mathcal{R}}$ are WSkS definable for left-linear, right ground TRSs over finite signatures. This suggests us to use the following approximation of left-linear right-ground TRSs. Given $\mathcal{R} = (\mathcal{F}, R)$, we define $\mathcal{R}_{rg} = (\mathcal{F} \cup \{\Omega\}, R_{rg})$ as follows:

$$R_{rg} = \{l \to r_\Omega \mid l \to r \in R\} \cup \{\Omega \to f(\overline{\Omega}) \mid f \in \mathcal{F}\}$$

where t_Ω is the term t with all variables replaced by Ω. It is straightforward to see that rg is an approximation of TRSs. The following theorem ensures that S-determinacy w.r.t. f and i is decidable for an approximation $\mathcal{R}_{rg}$ of a TRS $\mathcal{R}$ and semantics $\mathsf{S} \in \{\mathsf{nf}_{\mathcal{R}_{rg}}, \mathsf{eval}_{\mathcal{R}_{rg}}\}$.

Theorem 8
Let $\mathcal{R}$ be a left-linear TRS, $\mathcal{R}_{rg}$ be the approximation rg of $\mathcal{R}$, $f \in \mathcal{F}$, $i \in \{1, \ldots, ar(f)\}$, and $\mathsf{S} \in \{\mathsf{eval}_{\mathcal{R}_{rg}}, \mathsf{nf}_{\mathcal{R}_{rg}}\}$. It is decidable whether S is determined w.r.t. f and i.

By Theorems 6 and 8, redundancy of an argument w.r.t. $\mathsf{nf}_{\mathcal{R}}$ (and $\mathsf{eval}_{\mathcal{R}}$) is *effectively* approximable by using rg.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

Corollary 1 (*Approximation of redundancy for* $\mathscr{R}_{rg}$)
Let $\mathscr{R} = (\mathscr{F}, R)$ be a left-linear TRS, $f \in \mathscr{F}$, $i \in \{1, \ldots, ar(f)\}$, and $\mathsf{S} \in \{\mathsf{eval}, \mathsf{nf}\}$. If $\mathscr{R}$ is $\mathsf{S}_{\mathscr{R}}$-defined and $\mathsf{S}_{\mathscr{R}_{rg}}$ is determined w.r.t. f and i, then $i \in rarg_{\mathsf{S}_{\mathscr{R}}}(f)$.

Example 3
Consider the left-linear TRS $\mathscr{R}$

```
f(x,0) = 0          f(0,s(y)) = s(0)      f(s(x),s(y)) = g(x,y)
g(x,y) = f(x,s(y))
```

Note that $\mathscr{R}$ is terminating, hence $\mathsf{nf}_{\mathscr{R}}$-defined. Approximation $\mathscr{R}_{rg}$ is:

```
f(x,0) = 0          f(0,s(y)) = s(0)      f(s(x),s(y)) = g(Ω,Ω)
g(x,y) = f(Ω,s(Ω))  Ω = f(Ω,Ω)            Ω = g(Ω,Ω)
Ω = s(Ω)            Ω = 0
```

It is not difficult to see that $\mathsf{nf}_{\mathscr{R}_{rg}}$ is determined w.r.t. `f` and 1 whereas is not determined w.r.t. `f` and 2. It is possible to construct an automaton which tests those conditions, see Thatcher and Wright (1968) for more details, thus making it automatically provable. By Theorem 6, this means that $1 \in rarg_{\mathsf{nf}_{\mathscr{R}}}(\mathtt{f})$.

The approximation *rg* is similar to *nv* of (Durand and Middeldorp 1997), that replaces every variable in rhs's by fresh ones. However, including the new symbol Ω in the rhs's of the approximated program is essential for our development since the semantics of the program obtained by the approximation *nv* is not expressible in the logic WS*k*S.

In the following section, 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.

6 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.

Definition 3 (*p-prefix-equal terms*)
We say that two terms $t, s \in \mathscr{T}(\mathscr{F}, \mathscr{X})$ are *p-prefix-equal*, with $p \in \mathscr{P}os(t) \cap \mathscr{P}os(s)$ if, for all occurrences w with $w < p$, $t|_w$ and $s|_w$ have the same symbol at the root.

Definition 4 (*Redundant position*)
Let S be a term semantics for a signature $\mathscr{F}$ and $t \in \mathscr{T}(\mathscr{F}, \mathscr{X})$. The position $p \in \mathscr{P}os(t)$ is *redundant* in t w.r.t. S if, for all $t', s \in \mathscr{T}(\mathscr{F})$ such that t and t' are p-prefix-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 $s \in \mathscr{T}(\mathscr{F})$, $\mathsf{S}(t) = \mathsf{S}(t[s]_p)$, mimicking Definition 1. The

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

reason is that positions in a term cannot be analyzed independently for redundancy if we want our notion of redundancy of positions to be truly compositional, as the following example shows.

Example 4
Let us consider the TRS $\mathscr{R}$:

```
f(a,a) = a      f(a,b) = a      f(b,a) = a      f(b,b) = b
```

Given the term t = `f(a,a)`, for all terms $s \in \mathscr{T}(\mathscr{F})$, $\mathsf{eval}_{\mathscr{R}}(t[s]_1) = \mathsf{eval}_{\mathscr{R}}(t)$ and $\mathsf{eval}_{\mathscr{R}}(t[s]_2) = \mathsf{eval}_{\mathscr{R}}(t)$. However, $\mathsf{eval}_{\mathscr{R}}(t[\mathtt{b}]_1[\mathtt{b}]_2) \neq \mathsf{eval}_{\mathscr{R}}(t)$. Indeed, $1,2 \notin rpos_{\mathsf{eval}_{\mathscr{R}}}(t)$.

In the following, we extend Theorem 1 and Proposition 1 (which concern redundant arguments of function symbols) to redundant positions of terms.

Theorem 9 (*Antimonotonicity of redundancy of a position*)
Let S, S' be term semantics for a signature $\mathscr{F}$. If $\mathsf{S} \preceq \mathsf{S}'$, then, for all $t \in \mathscr{T}(\mathscr{F}, \mathscr{X})$, $rpos_{\mathsf{S}'}(t) \subseteq rpos_{\mathsf{S}}(t)$.

Proposition 3 (*Non-redundancy of constructor positions*)
Let $\mathscr{R}$ be a TRS such that $|\mathscr{T}(\mathscr{C})| > 1$, and S be a rewriting semantics such that $\mathsf{eval}_{\mathscr{R}} \preceq \mathsf{S}$. Then, for all $t \in \mathscr{T}(\mathscr{C}, \mathscr{X})$, $rpos_{\mathsf{S}}(t) = \emptyset$.

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 4
Let S be a term semantics for a signature $\mathscr{F}$, $t \in \mathscr{T}(\mathscr{F}, \mathscr{X})$, $p \in \mathscr{P}os(t)$, $f \in \mathscr{D}$. 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, 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. Theorems 1 and 9 say 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.

6.1 Using Redundant Positions for Characterizing Redundancy: the Variable Case

In this section, we focus on the problem of characterizing the redundant arguments w.r.t. $\mathsf{eval}_{\mathscr{R}}$ by studying the redundancy w.r.t. $\mathsf{eval}_{\mathscr{R}}$ of some positions in the rhs's of program rules. The following definition is useful to detect whether the variables of the i-th argument in a lhs of symbol f propagate to positions in the rhs under the same i-th argument of symbol f.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

Definition 5 (*(f,i)-redundant variable*)
Let $f \in \mathscr{D}$, $i \in \{1,\ldots,ar(f)\}$, and $t \in \mathscr{T}(\mathscr{F},\mathscr{X})$. The variable $x \in \mathscr{X}$ is *(f,i)-redundant in t* if it occurs only at positions $p \in \mathscr{P}os_x(t)$ which (i) are redundant w.r.t. $\mathsf{eval}_{\mathscr{R}}$ in t, i.e., $p \in rpos_{\mathsf{eval}_{\mathscr{R}}}(t)$, or (ii) they appear inside the i-th parameter of f-rooted subterms of t, i.e., $\exists q$ such that $q.i \leqslant p$ and $root(t|_q) = f$.

Note that variables which do not occur in a term t are trivially (f,i)-redundant in t for any $f \in \mathscr{F}$ and $i \in \{1,\ldots,ar(f)\}$.

Example 5
Consider the rules for symbol `lastnew` in Example 1:

```
lastnew(x,nil,z) = z        lastnew(x,y:ys,z) = lastnew(y,ys,z)
```

Variable `x` is (`lastnew`, 1)-redundant in rhs's r_1 = `z` and r_2 = `lastnew(y,ys,z)`, since it does not appear in them. Variable `ys` is (`lastnew`, 2)-redundant in rhs r_2, since it appears under the second argument of symbol `lastnew`.

Now, we are able to provide the second effective method to determine redundant arguments based on the (f,i)-redundant variables occurring in rhs's. In order to prove Theorem 10 below, we introduce some auxiliary definitions and lemmata.

Given a TRS $\mathscr{R} = (\mathscr{F},R)$, we write $\mathscr{R}_f$ to denote the TRS $\mathscr{R}_f = (\mathscr{F},\{l \to r \in R \mid root(l) = f\})$ which contains the set of rules defining $f \in \mathscr{D}$. The following definition provides the set of positions of the i-th parameter of f symbols in t.

Definition 6
Let $f \in \mathscr{F}$, $i \in \{1,\ldots,ar(f)\}$, and $t \in \mathscr{T}(\mathscr{F},\mathscr{X})$. We define $Pos_{f,i}(t) = \{q.i \in \mathscr{P}os(t) \mid root(t|_q) = f\}$.

Let $\bar{t} = t_1,\ldots,t_n$ be a sequence of terms, $P = p_1,\ldots,p_n$ be a sequence of positions of another term s, and $P' = p'_1,\ldots,p'_m$ be a subsequence of P (i.e., $m < n$ and $\exists \mu : \{1,\ldots,m\} \to \{1,\ldots,n\}$ such that $p'_i = p_{\mu(i)}$ and $i < i' \Rightarrow \mu(i) < \mu(i')$), we denote $\bar{t}|_{P'} = t'_1,\ldots,t'_m$ such that $t'_i = t_{\mu(i)}$. The following result is auxiliary and proves that the same constructor term is obtained by rewriting when we replace the set of subterms at $\mathsf{eval}_{\mathscr{R}}$-redundant and $Pos_{f,i}$ positions in a term by an arbitrary set of terms.

Proposition 5
Let $\mathscr{R}$ be a left-linear CS, $f \in \mathscr{D}$, and $i \in \{1,\ldots,ar(f)\}$. Let $t \in \mathscr{T}(\mathscr{F})$, $P \subseteq Pos_{f,i}(t) \cup rpos_{\mathsf{eval}_{\mathscr{R}}}(t)$ be a set of disjoint positions, and $\bar{s} \in \mathscr{T}(\mathscr{F})$. Let $t \to^* \delta$ for some $\delta \in \mathscr{T}(\mathscr{C})$. If, for all $l \to r \in \mathscr{R}_f$, $l|_i$ is a variable which is (f,i)-redundant in r, then $t[\bar{s}]_P \to^* \delta$.

Now, we provide the second effective method to detect redundancy.

Theorem 10 (*Detecting redundancy: the Variable Case*)
Let $\mathscr{R}$ be a left-linear CS. Let $f \in \mathscr{D}$ and $i \in \{1,\ldots,ar(f)\}$. If, for all $l \to r \in \mathscr{R}_f$, $l|_i$ is a variable which is (f,i)-redundant in r, then $i \in rarg_{\mathsf{eval}_{\mathscr{R}}}(f)$.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

Example 6

A standard example in the literature on *useless variable elimination* (UVE) – a popular technique for removing dead variables, see Wand and Siveroni (1999) and Kobayashi (2000) – is the following program[4] with constructor symbols $\mathscr{C} = \{0, s\}$ and variables a, bogus, and j:

```
loop(a,bogus,0) = loop(s(a),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 10, the second argument of loop is redundant w.r.t. $\mathsf{eval}_{\mathscr{R}}$.

The restriction to left-linear rules in Theorem 10 above is not strictly necessary; however, in most practical cases the redundancy of the argument of symbol f cannot be analyzed independently when we consider repeated variables in left-hand sides, as witnessed by the following example.

Example 7

Consider the TRS $\mathscr{R}$:

```
f(x,x) = a
```

where f and a are the only function symbols in the signature. Since every ground term t rewrites to a (this can be easily proved by structural induction), both arguments of f are redundant w.r.t. $\mathsf{eval}_{\mathscr{R}}$. However, if we add a new constant symbol b, then no argument of f is redundant anymore.

The following example demonstrates that the restriction to constructor systems in Theorem 10 is also necessary.

Example 8

Consider the following non-constructor TRS $\mathscr{R}$ where $\mathscr{C} = \{a, b\}$:

```
f(a,x) = g(f(b,x))      g(f(b,x)) = x
```

Then, the second argument of f(a,x) in the lhs of the first rule is a variable which, in the corresponding rhs of the rule, occurs within the second argument of a subterm rooted by f, namely f(b,x). Hence, by Theorem 10 we would have that $2 \in rarg_{\mathsf{eval}_{\mathscr{R}}}(\mathtt{f})$. However, $\mathsf{eval}_{\mathscr{R}}(\mathtt{f(a,a)}) = \{\mathtt{a}\} \neq \{\mathtt{b}\} = \mathsf{eval}_{\mathscr{R}}(\mathtt{f(a,b)})$, which contradicts $2 \in rarg_{\mathsf{eval}_{\mathscr{R}}}(\mathtt{f})$.

Moreover, the extension of this result to the normalization semantics nf is not possible, as shown in the following example.

Example 9

Consider the TRS $\mathscr{R}$ where $\mathscr{C} = \{a, b\}$:

```
f(a,x) = a
```

[4] 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 only with two rules.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

This TRS satisfies the conditions of Theorem 10 and then $2 \in rarg_{\mathsf{eval}_{\mathscr{R}}}(\mathtt{f})$. In concrete, we have that, for all s, $\mathsf{eval}_{\mathscr{R}}(\mathtt{f(b,}s\mathtt{)}) = \emptyset$. However, $\mathsf{nf}_{\mathscr{R}}(\mathtt{f(b,a)}) = \{\mathtt{f(b,a)}\} \neq \{\mathtt{f(b,b)}\} = \mathsf{nf}_{\mathscr{R}}(\mathtt{f(b,b)})$.

Now, we are able to detect some redundancies in Example 1.

Example 10
Let us revisit the following rules from the CS $\mathscr{R}$ of Example 1:

```
lastnew(x,nil,z) = z          lastnew(x,y:ys,z) = lastnew(y,ys,z)
```

Using Theorem 10, we are able to conclude that the first argument of function `lastnew` is (trivially) redundant w.r.t. $\mathsf{eval}_{\mathscr{R}}$, since, in every lhs, the first parameter of `lastnew` is a variable that is (`lastnew`, 1)-redundant in the respective rhs.

Unfortunately, Theorem 10 does not suffice to prove that the *second* argument of `lastnew` is redundant w.r.t. $\mathsf{eval}_{\mathscr{R}}$, and this motivates the next section.

6.2 Using redundant positions for characterizing redundancy: the pattern case

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 confluence and $\mathsf{eval}_{\mathscr{R}}$-definedness, in return. The following definitions are helpful to determine the redundancy of argument i of f when f is defined by 'matching cases' for the argument i in the different rules.

Definition 7
Let $\mathscr{F}$ 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 8 ((f,i)-*triple*)
Let $\mathscr{R} = (\mathscr{F},R)$ be a TRS, $f \in \mathscr{F}$, and $i \in \{1,\ldots,ar(f)\}$. Given two different (possibly renamed) rules $l \to r$, $l' \to r'$ in $\mathscr{R}_f$ such that $\mathscr{V}ar(l) \cap \mathscr{V}ar(l') = \emptyset$, we say that $\langle l \to r, l' \to r', \sigma\rangle$ is an (f,i)-triple of $\mathscr{R}$ if l and l' unify up to i-th argument with *mgu* σ.

Example 11
Consider the TRS $\mathscr{R}$ from Example 1. This program has a single (`lastnew`, 2)-triple:

$\langle$`lastnew(x,nil,z)=z`, `lastnew(x',y:ys,z')=lastnew(y,ys,z')`, $[\mathtt{x} \mapsto \mathtt{x'}, \mathtt{z} \mapsto \mathtt{z'}]\rangle$

The following definition allows us to consider rules for symbol f which are "semantically equivalent" after replacing some variables and i-parameters in their rhs's. The basic idea is to check joinability of the (f,i)-triples of Definition 8 where variables below the i-th argument of symbol f in the left-hand sides of the rules of the triple are explicitly instantiated by a *dummy* symbol a (Definition 9 below). Intuitively, joinabilty of (all) such triples, then, amounts at proving the i-th argument of f as redundant (Theorem 11 below).

In the following, we will use notation $\bar{t}$ either for a k-tuple of terms $t_1,\ldots,t_k$ or for a sequence of a unique term $t,\ldots,t$; the distinction will be clarified by the context.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

Definition 9 (*Joinable* (f,i)*-triple*)
Let $\mathcal{R}$ be a TRS, $f \in \mathcal{D}$, and $i \in \{1,\ldots,ar(f)\}$. Let a be an arbitrary constant. An (f,i)-triple $\langle l \to r, l' \to r', \sigma\rangle$ of $\mathcal{R}$ is joinable if $\sigma_{\mathscr{C}}(\tau_l(r))$ and $\sigma_{\mathscr{C}}(\tau_{l'}(r'))$ are joinable (i.e., they have a common reduct). Here, substitution $\sigma_{\mathscr{C}}$ is given by:

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

and transformation τ_l is given by

$$\tau_l(t) = \begin{cases} t & \text{if } l|_i \in \mathcal{X} \\ t[\overline{a}]_Q & \text{if } l|_i \notin \mathcal{X} \text{ and } Q = \{p \in Pos_{f,i}(t) \mid \mathcal{V}ar(t|_p) \cap \mathcal{V}ar(l|_i) \neq \emptyset\} \end{cases}$$

Note that the constant a in the previous definition can be replaced by any ground term. In the case of many-sorted signatures, we would consider different constants 'a', one for each sort.

Example 12
Consider again the CS $\mathcal{R}$ in Example 1 and the single (`lastnew`, 2)-triple given in Example 11. Let us call the rhs's

$$r_1 = \texttt{z} \text{ and } r_2 = \texttt{lastnew(y,ys,z')}$$

for the lh's $l_1 =$ `lastnew(x,nil,z)` and $l_2 =$ `lastnew(x',y:ys,z')`. Let us consider that `0` is the constant for the sort of the first argument of `lastnew` and `nil` is the constant for the sort of the second argument of `lastnew`. The corresponding transformed rhs's are

$$\tau_{l_1}(r_1) = \texttt{z} \text{ and } \tau_{l_2}(r_2) = \texttt{lastnew(y,nil,z')}.$$

With $\sigma = [\texttt{x} \mapsto \texttt{x'}, \texttt{z} \mapsto \texttt{z'}]$ and $\sigma_{\mathscr{C}} = [\texttt{x} \mapsto \texttt{x'}, \texttt{z} \mapsto \texttt{z'}, \texttt{y} \mapsto \texttt{0}, \texttt{ys} \mapsto \texttt{nil}]$, the corresponding instantiated rhs's are

$$\sigma_{\mathscr{C}}(\tau_{l_1}(r_1)) = \texttt{z'} \text{ and } \sigma_{\mathscr{C}}(\tau_{l_2}(r_2)) = \texttt{lastnew(0,nil,z')}.$$

We can prove $\sigma_{\mathscr{C}}(\tau_{l_1}(r_1))$ and $\sigma_{\mathscr{C}}(\tau_{l_2}(r_2))$ are joinable, since the variable `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. $\mathsf{eval}_{\mathcal{R}}$ which is based on finding a common reduct of (some particular instances of) the right-hand sides of rules.

Definition 10 ((f,i)*-joinable TRS*)
Let $\mathcal{R}$ be a TRS, $f \in \mathcal{F}$, 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.

The following result is auxiliary for Theorem 11 and proves that the same constructor term is obtained by rewriting when we replace the set of subterms at $\mathsf{eval}_{\mathcal{R}}$-redundant and $Pos_{f,i}$ positions in a term by an arbitrary set of terms.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

Proposition 6
Let $\mathscr{R}$ be a left-linear, confluent, and $\mathsf{eval}_{\mathscr{R}}$-defined CS. Let $f \in \mathscr{D}$ and $i \in \{1,\ldots,ar(f)\}$. Let $t \in \mathscr{T}(\mathscr{F})$, $P \subseteq Pos_{f,i}(t) \cup rpos_{\mathsf{eval}_{\mathscr{R}}}(t)$ be a set of disjoint positions, and a be a constant. Let $t \rightarrow^* \delta$ for some $\delta \in \mathscr{T}(\mathscr{C})$. If $\mathscr{R}$ is (f,i)-joinable, then $t[\overline{a}]_P \rightarrow^* \delta$.

Now, we provide the third effective method to detect redundancy.

Theorem 11 (*Detecting redundancy: the Pattern Case*)
Let $\mathscr{R}$ be a left-linear, confluent and $\mathsf{eval}_{\mathscr{R}}$-defined CS. Let $f \in \mathscr{D}$ and $i \in \{1,\ldots,ar(f)\}$. If $\mathscr{R}$ is (f,i)-joinable, then $i \in rarg_{\mathsf{eval}_{\mathscr{R}}}(f)$.

Confluence and $\mathsf{eval}_{\mathscr{R}}$-definedness are necessary, as shown in the following examples.

Example 13
Consider the following non-confluent CS $\mathscr{R}$:

```
f(0) = 0        f(s(x)) = g(f(x))        g(x) = 0        g(x) = s(0)
```

By Theorem 11, we would have $1 \in rarg_{\mathsf{eval}_{\mathscr{R}}}(\mathtt{f})$, since the $(\mathtt{f},1)$-triple $\langle$`f(0)=0`, `f(s(x))=g(f(x))`, *id*$\rangle$ is joinable, i.e., the common reduct of terms `0` and `g(f(0))` is `0`. However, $\mathsf{eval}_{\mathscr{R}}(\mathtt{f(0)}) = \{\mathtt{0}\} \neq \{\mathtt{0},\mathtt{s(0)}\} = \mathsf{eval}_{\mathscr{R}}(\mathtt{f(s(0))})$.

Example 14
Consider the following non-$\mathsf{eval}_{\mathscr{R}}$-defined CS $\mathscr{R}$:

```
f(0) = 0        f(s(x)) = f(x)        g(s(0)) = 0
```

By Theorem 11, we would have $1 \in rarg_{\mathsf{eval}_{\mathscr{R}}}(\mathtt{f})$. But $\mathsf{eval}_{\mathscr{R}}(\mathtt{f(0)}) = \{\mathtt{0}\} \neq \emptyset = \mathsf{eval}_{\mathscr{R}}(\mathtt{f(g(0))})$.

Joinability is decidable for terminating, confluent TRSs as well as for other classes of TRSs such as right-ground TRSs (Oyamaguchi 1990) and confluent semi-constructor TRSs (Mitsuhashi *et al.* 2004) (a semi-constructor TRS is such a TRS that every subterm of the rhs of each rewrite rule is ground if its root is a defined symbol). Hence, Theorem 11 gives us an effective method to recognize redundancy in completely defined, confluent, and (semi-)complete TRSs, as illustrated in the following.

Example 15
Consider again the CS $\mathscr{R}$ of Example 1. This program is confluent, terminating and completely defined (considering sorts), hence is $\mathsf{eval}_{\mathscr{R}}$-defined. By Example 10, the first argument of `lastnew` is redundant w.r.t. $\mathsf{eval}_{\mathscr{R}}$, using Theorem 10. Now, the second argument of `lastnew` is redundant w.r.t. $\mathsf{eval}_{\mathscr{R}}$ using the new Theorem 11. 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`, $\mathscr{R}_{\mathtt{applast}}$ is $(\mathtt{applast},1)$-joinable. And again by Theorem 11, we conclude that the first argument of `applast` is also redundant. Hence, $1 \in rarg_{\mathsf{eval}_{\mathscr{R}}}(\mathtt{applast})$ and $1,2 \in rarg_{\mathsf{eval}_{\mathscr{R}}}(\mathtt{lastnew})$.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

Table 1. *Summary of Results*

Semantics	Theorem	Requirements
S	Th. 1 (Antimonotonicity)	–
S	Prop. 1 (Non-redundancy)	–
{red, hnf, nf, eval}	Th. 5 (Decidability)	LL, RG
{nf, eval}	Coro. 1 (Approximation $\mathscr{R}_{rg}$)	LL, ND, NDT (ED, EDT)
S	Th. 9 (Antimonotonicity – positions)	–
S	Prop. 3 (Non-redundancy – positions)	–
eval	Th. 10 (The Variable Case)	CS, LL, VR
eval	Th. 11 (The Pattern Case)	C, CS, ED, LL, JT

C	: Confluence	LL	: Left-Linearity of the TRS
CS	: Constructor System	ND	: nf-definedness
ED	: eval-definedness	NDT	: $\mathsf{nf}_{\mathscr{R}_{rg}}$-determinacy
EDT	: $\mathsf{eval}_{\mathscr{R}_{rg}}$-determinacy	RG	: Right-ground TRS
JT	: Joinability of (f,i)-triples	VR	: Variables in $l\|_i$ are (f,i)-redundant in r

Let us conclude with a few general remarks about the complexity of our approach, that is, the analysis time to detect redundant arguments (the cost of performing the optimizations proposed in Section 7 is negligible). In Table 1, we provide a summary of the main results in the paper. Theorem 10 only requires syntactic properties which can be tested in linear time on the size of the TRS (i.e., on the sum of sizes of each rule, where the size of a rule is the sum of sizes of the left- and right-hand sides). The conditions LL, C and ED in the premises of Theorem 11 are standard properties of rewrite systems (as remarked in Section 3, a TRS $\mathscr{R}$ is ED if $\mathscr{R}$ is normalizing and completely defined, but there is no direct way to check whether a TRS is normalizing and then termination is required) and then assumed to be fulfilled by the TRS $\mathscr{R}$ and checked apart. The complexity of such properties for decidable cases has been investigated eg (Godoy and Tiwari (2004); Kapur *et al.* (1987); Verma (2002) and a number of tools are available for checking them in practice: For instance, termination tools such as AProVE (Giesl *et al.* 2004) and C*i*ME (Contejean *et al.* 2003), confluence checking tools such as C*i*ME, and tools for ensuring completely-definedness such as Scc (Hendrix *et al.* 2005). Thus, the only property which is strictly new in our framework is JT. As we mentioned above, joinability is decidable for several classes of TRSs (Godoy and Tiwari 2004; Verma 2002; Mitsuhashi *et al.* 2004). Actually, there are (cubic) polynomial time algorithms for joinability of ground systems (Verma 2002, Theorem 12) and a slightly more general class of TRSs is considered in (Godoy and Tiwari 2004), namely right-(ground or variable) rewrite systems. In our implementation however, confluence and termination of the TRS are assumed for the application of Theorem 11 (see above) and then joinability of terms t and s is decidable by just checking whether the normal forms of t and s are equal.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

7 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. Then, since redundant arguments are not necessary to determine the result of a function call, it is worth to develop methods and techniques to avoid such unpleasant effects.

As remarked in the introduction, inefficiencies caused by the redundancy of arguments cannot (in general) be avoided by using rewriting strategies. 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 $\mathscr{R}$. Hence, we remove redundant formal parameters and corresponding actual parameters for each function symbol and function call in $\mathscr{R}$. We begin with the notion of syntactic erasure which is intended to pick up redundant arguments of function symbols.

Definition 11 (*Syntactic erasure*)
A *syntactic erasure* is a mapping $\rho : \mathscr{F} \to \mathscr{P}(\mathbb{N})$ such that for all $f \in \mathscr{F}$, $\rho(f) \subseteq \{1, \ldots, ar(f)\}$. We say that a syntactic erasure ρ is *sound* for a semantics S if, for all $f \in \mathscr{F}$, $\rho(f) \subseteq rarg_{\mathsf{S}}(f)$.

Example 16
Given the signature $\mathscr{F} = \{\mathtt{0}, \mathtt{nil}, \mathtt{s}, \mathtt{:}, \mathtt{applast}, \mathtt{lastnew}\}$ of the TRS $\mathscr{R}$ in Example 1, with $ar(\mathtt{0}) = ar(\mathtt{nil}) = 0$, $ar(\mathtt{s}) = 1$, $ar(\mathtt{:}) = ar(\mathtt{applast}) = 2$, and $ar(\mathtt{lastnew}) = 3$, and according to Example 15, the following mapping ρ is a *sound syntactic erasure* for the semantics $\mathsf{eval}_{\mathscr{R}}$: $\rho(\mathtt{0}) = \rho(\mathtt{nil}) = \rho(\mathtt{s}) = \rho(\mathtt{:}) = \emptyset$, $\rho(\mathtt{applast}) = \{1\}$, and $\rho(\mathtt{lastnew}) = \{1, 2\}$.

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 12 (*Erasure of a signature*)
Given a signature $\mathscr{F}$ and a syntactic erasure $\rho : \mathscr{F} \to \mathscr{P}(\mathbb{N})$, the erasure of $\mathscr{F}$ is the signature $\mathscr{F}_\rho$ whose symbols $f_\rho \in \mathscr{F}_\rho$ are one to one with symbols $f \in \mathscr{F}$ and whose arities are related by $ar(f_\rho) = ar(f) - |\rho(f)|$.

Example 17
The erasure of the signature in Example 16 is $\mathscr{F}_\rho = \{\mathtt{0}, \mathtt{nil}, \mathtt{s}, \mathtt{:}, \mathtt{applast}, \mathtt{lastnew}\}$, with $ar(\mathtt{0}) = ar(\mathtt{nil}) = 0$, $ar(\mathtt{s}) = ar(\mathtt{applast}) = ar(\mathtt{lastnew}) = 1$, and $ar(\mathtt{:}) = 2$. 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 13 (*Erasure of a term*)
Given a syntactic erasure $\rho : \mathscr{F} \to \mathscr{P}(\mathbb{N})$, the function $\tau_\rho : \mathscr{T}(\mathscr{F}, \mathscr{X}) \to \mathscr{T}(\mathscr{F}_\rho, \mathscr{X})$ on terms is: $\tau_\rho(x) = x$ if $x \in \mathscr{X}$ 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 $1 \leqslant m < k$.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

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 $\mathcal{F}$ (which automatically belongs to $\mathcal{F}_\rho$).

Definition 14 (*Erasure of a TRS*)
Let $\mathcal{R}$ be a TRS, a a constant, and ρ be a syntactic erasure for $\mathcal{F}$. The erasure $\mathcal{R}_\rho$ of $\mathcal{R}$ is $\mathcal{R}_\rho = (\mathcal{F}_\rho, \{\tau_\rho(l) \to \sigma_l(\tau_\rho(r)) \mid l \to r \in R\})$ 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 $y \in \mathcal{V}ar(\tau_\rho(l))$.

Note that the constant a in the previous definition can be replaced by any ground term. In a many-sorted signature, we will have different constants 'a', each one of an appropriate sort.

Example 18
Let $\mathcal{R}$ be the TRS of Example 1 and ρ be the sound syntactic erasure of Example 16. The erasure $\mathcal{R}_\rho$ of $\mathcal{R}$ consists of the erased signature of Example 17 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 at obtaining the final, "optimal" program.

The mapping τ_ρ induces an equivalence $\equiv_{\tau_\rho}$ on terms given by: $t \equiv_{\tau_\rho} s$ iff $\tau_\rho(t) = \tau_\rho(s)$. We have the following property of sound erasures of terms.

Proposition 7
If the syntactic erasure $\rho : \mathcal{F} \to \mathcal{P}(\mathbb{N})$ is sound with respect to the semantics S, then for all $t, s \in \mathcal{T}(\mathcal{F})$, $t \equiv_{\tau_\rho} s$ implies that $\mathsf{S}(t) = \mathsf{S}(s)$.

The following theorem establishes the correctness of the erasure procedure for a rewriting semantics S.

Theorem 12 (*Correctness*)
Let $\mathcal{R}$ be a left-linear TRS, S be a rewriting semantics for $\mathcal{R}$, ρ be a sound syntactic erasure for S, and $t \in \mathcal{T}(\mathcal{F})$. If $\delta \in \mathsf{S}(t)$, then $\tau_\rho(t) \to^*_{\mathcal{R}_\rho} \tau_\rho(\delta)$.

The following theorem establishes the completeness of the erasure procedure for a rewriting semantics S.

Theorem 13 (*Completeness*)
Let $\mathcal{R}$ be a left-linear TRS, S be a rewriting semantics for $\mathcal{R}$ such that $\mathsf{S} \preceq \mathsf{red}_{\mathcal{R}}$, ρ be a sound syntactic erasure for S, and $t, \delta \in \mathcal{T}(\mathcal{F}_\rho)$. If $t \to^*_{\mathcal{R}_\rho} \delta$, then $\forall t', \delta' \in \mathcal{T}(\mathcal{F})$ such that $\tau_\rho(t') = t$ and $\tau_\rho(\delta') = \delta$, $\mathsf{S}(\delta') \subseteq \mathsf{S}(t')$.

The following theorem establishes the correctness and completeness of the erasure procedure for the semantics $\mathsf{eval}_{\mathcal{R}}$.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

Theorem 14 (*Correctness and Completeness*)
Let $\mathscr{R}$ be a left-linear TRS, ρ be a sound syntactic erasure for $\mathsf{eval}_{\mathscr{R}}$, $t \in \mathscr{T}(\mathscr{F})$, and $\delta \in \mathscr{T}(\mathscr{C})$. Then, $\tau_\rho(t) \to^*_{\mathscr{R}_\rho} \delta$ iff $\delta \in \mathsf{eval}_{\mathscr{R}}(t)$.

In the following, we are able to ascertain the conditions for the preservation of some computational properties of TRSs after erasure.

Theorem 15 (*Preservation of Confluence*)
Let $\mathscr{R}$ be a left-linear TRS. Let ρ be a sound syntactic erasure for $\mathsf{eval}_{\mathscr{R}}$. If $\mathscr{R}$ is $\mathsf{eval}_{\mathscr{R}}$-defined and confluent, then the erasure $\mathscr{R}_\rho$ of $\mathscr{R}$ is confluent.

Theorem 16 (*Preservation of Normalization*)
Let $\mathscr{R}$ be a left-linear and completely defined TRS, and ρ be a sound syntactic erasure for $\mathsf{eval}_{\mathscr{R}}$. If $\mathscr{R}$ is normalizing, then the erasure $\mathscr{R}_\rho$ of $\mathscr{R}$ is normalizing.

In the theorem above, we cannot strengthen normalization to termination. A simple counterexample showing that termination may get lost is the following; note that the opposite is also possible, i.e., a non-terminating TRS can be made terminating after the erasure.

Example 19
Consider the left-linear, (confluent, completely defined, and) terminating TRS $\mathscr{R}$

```
h(a,y) = a      h(c(x),y) = h(x,c(y))
```

The first argument of `h` is redundant w.r.t. $\mathsf{eval}_{\mathscr{R}}$. However, after erasing the argument, we get the TRS

```
h(y) = a        h(y) = h(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.

After the erasure, a post-processing transformation able to remove redundant rules (w.r.t. an appropriate notion of rule redundancy) might be useful to restore termination or orthogonality in some cases, as the example above. Although this point is outside the scope of this paper, in the following we provide a program transformation that can improve the optimization achieved by the erasure.

Definition 15 (*Reduced erasure of a TRS*)
Let $\mathscr{R}$ be a TRS and ρ be a syntactic erasure for $\mathscr{F}$. The reduced erasure $\mathscr{R}'_\rho$ of $\mathscr{R}$ is obtained from the erasure $\mathscr{R}_\rho$ of $\mathscr{R}$ by a *compression transformation* defined as removing any trivial rule $t \to t$ of $\mathscr{R}_\rho$ and then normalizing the rhs's of the rules w.r.t. the non-trivial rules of $\mathscr{R}_\rho$.

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

Example 20
Let $\mathscr{R}_\rho$ be the erasure of Example 18. The reduced erasure consists of the rules {`applast(z) = z, lastnew(z) = z`}.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

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) (Gramlich 2001), and the removal of trivial rules does not change the evaluation semantics of the TRS $\mathscr{R}$ either, we have the following.

Corollary 2
Let $\mathscr{R}$ be a left-linear TRS, ρ be a sound syntactic erasure for $\mathsf{eval}_{\mathscr{R}}$, $t \in \mathscr{T}(\mathscr{F})$, and $\delta \in \mathscr{T}(\mathscr{C})$. If (the TRS which results from removing trivial rules from) $\mathscr{R}_\rho$ is confluent and terminating (alternatively, if it is almost orthogonal and normalizing), then, $\tau_\rho(t) \to^*_{\mathscr{R}'_\rho} \delta$ if and only if $\delta \in \mathsf{eval}_{\mathscr{R}}(t)$, where $\mathscr{R}'_\rho$ is the reduced erasure of $\mathscr{R}$.

Erasures and reduced erasures of a TRS preserve left-linearity. For a TRS $\mathscr{R}$ satisfying the conditions in Corollary 2, by using Gramlich (2001), it is immediate that the reduced erasure $\mathscr{R}'_\rho$ is confluent and normalizing. Also, $\mathscr{R}'_\rho$ is completely defined if $\mathscr{R}$ is.

Hence, let us note that these results allow us to perform the 'optimal' optimization of program `applast` in Example 1 while guaranteeing that the intended (evaluation or normalization) semantics is preserved.

8 Experiments

The practicality of our ideas is witnessed by the implementation of a prototype system which delivers encouraging good results for the techniques deployed in Section 6 (Theorems 10 and 11) and the erasure procedure of Section 7. The prototype has been implemented in PAKCS (Hanus *et al.* 2003a), the current distribution[5] of the multi-paradigm declarative language Curry (Hanus *et al.* 2003b), and is publicly available at `http://www.dsic.upv.es/users/elp/redargs`.

We have used the prototype to perform some preliminary experiments which show that our methodology does detect and remove redundant arguments of some common transformation benchmarks, such as `bogus`, `lastappend`, `allzeros`, `doubleflip`, etc.; see (Leuschel 1998) and references therein. Tables 2 and 3 summarize the experiments. Benchmarks code as well as the programs obtained by the erasure procedure are included in (Alpuente *et al.* 2006).

Table 2 shows the execution runtimes of the original and transformed programs in PAKCS, as well as the arguments in the whole program which are signaled as redundant for each benchmark using the notation: #signaled/#total. Runtimes have been measured in an "AMD Athlon XP" class machine running Fedora Core 3.0 and using version 1.6.0 of the PAKCS compiler under SICStus Prolog 3.8.6. Natural numbers are given by numbers 0, 1, 2, etc in the tables, instead of the notation `Z/S` x used in the bechmarks code. For benchmarking purposes, goals make use of the auxiliary factorial function, defined in a usual way. The number of elements of a list (when used) is indicated by a subindex. Note that the analysis time for each example is negligible.

[5] See `http://www.informatik.uni-kiel.de/~pakcs`

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

Table 2. *Execution of the original and transformed programs in* Curry

Name	Call in original/erased program	Time (ms)	Gain	$rarg_{\text{eval}}$
bogus	`loop (fact 8) (fact 9) (fact 8)`	150		
	`loop' (fact 8) (fact 8)`	150	0%	1/1
applast	`applast` $[$`(fact 8)`$]_{10000}$ `(fact 8)`	168		
	`applast' (fact 8)`	153	9%	3/3
plus_minus	`minus_pe (fact 8) (fact 8)`	220		
	`minus_pe' (fact 8)`	155	30%	1/1
plus_leq	`leq_pe (fact 8) (fact 8)`	79		
	`leq_pe'`	∼0	100%	1/1
double_even	`even_pe (fact 8)`	77		
	`even_pe'`	∼0	100%	1/1
sum_allzeros	`sum_pe` $[$`(fact 8)`$]_{10000}$	23		
	`sum_pe'`	∼0	100%	1/1
Mutual recursion 1	`f (fact 8) (fact 8)`	123		
	`f'`	∼0	100%	1/1
Mutual recursion 2	`f (fact 8)`	132		
	`f'`	∼0	100%	1/1

Important optimizations are obtained for most examples. In the case of program `bogus`, no appreciable optimization is achieved by removing redundant arguments, since Curry is a lazy language and the redundant argument in `bogus` is a useless variable. In order to dissociate the possible dependency of the achieved optimization w.r.t. the lazy evaluation of the language, Table 3 shows the execution runtimes of the benchmarks in the Maude interpreter[6] (version 2.1.1), which uses an innermost rewriting strategy.

Note that, in this case, significant optimizations are also measured for programs `bogus` and `applast`. The `plus_minus` example runs in nearly half the original execution time in both, lazy and eager systems, which seems consistent with the fact that one of the two arguments have been removed.

9 Related work

Some notions have appeared in the literature of what it means for a term in a TRS $\mathscr{R}$ to be "computationally irrelevant". As we are going to see, 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 Kuper (1994) and Kennaway *et al.* (1996) is a property of the terms themselves (they may have

[6] See `http://maude.cs.uiuc.edu`

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

Table 3. *Execution of the original and transformed programs in* Maude

Name	Call in original/erased program	Time (ms)	Gain	$rarg_{\mathsf{eval}}$
bogus	`loop (fact 8) (fact 9) (fact 8)`	651		
	`loop' (fact 8) (fact 8)`	47	93%	1/1
applast	`applast` $[$`(fact 8)`$]_{10000}$ `(fact 8)`	102		
	`applast' (fact 8)`	54	47%	3/3
plus_minus	`minus_pe (fact 8) (fact 8)`	62		
	`minus_pe' (fact 8)`	30	51%	1/1
plus_leq	`leq_pe (fact 8) (fact 8)`	33		
	`leq_pe'`	∼0	100%	1/1
double_even	`even_pe (fact 8)`	32		
	`even_pe'`	∼0	100%	1/1
sum_allzeros	`sum_pe` $[$`(fact 8)`$]_{10000}$	40		
	`sum_pe'`	∼0	100%	1/1
Mutual recursion 1	`f (fact 8) (fact 8)`	73		
	`f'`	∼0	100%	1/1
Mutual recursion 2	`f (fact 8)`	61		
	`f'`	∼0	100%	1/1

meaning in $\mathscr{R}$ or may not), whereas our notion refers to arguments (positions) of function symbols. In (Kuper 1994, 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 Kuper (1994) 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 Kuper (1994) and Kennaway *et al.* (1996), whereas we might have redundant actual parameters 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 (Cousot and Cousot 1994; Hughes 1988), and the *filtering* of useless arguments and unnecessary variables of logic programming (Leuschel and Sørensen 1996; Pettorossi and Proietti 1994) are the closest to our work. In (Hughes 1988), 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 subterm 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

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

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, as shown in the following example.

Example 21
Consider the following TRS defining the length function for lists.

```
length(nil) = 0                    length(x:xs) = s(length(xs))
```

Hughes' analysis is able to determine that, in the `length` function, 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 Huet and Lévy (1991): 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 Cousot and Cousot (1994). Cousot's comportment analysis generalizes not only the unneededness analyses but also strictness, termination and other standard analyses of functional programming. In Cousot and Cousot (1994), 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 (Leuschel and Sørensen 1996), 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 in detail.

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 (Wand and Siveroni 1999) or a type-based approach (Berardi *et al.* 2000; Kobayashi 2000); see Berardi *et al.* (2000) 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

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

is Liu and Stoller (2002), 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 Liu and Stoller (2002) 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.

Example 22

Consider the TRS of Example 21 together with the following function symbol `f`:

```
f(x) = length(x:nil)
```

Our methods do not capture the redundancy of the argument of `f`. In Liu and Stoller (2002), it is shown that, in order to evaluate `length(xs)`, we do not need to evaluate the elements of the argument list `xs`; as Hughes's unneededness. In Liu et al.'s methodology, this means that we could replace the rule for `f` above by the rule `f(_) = length(_:nil)` where `_` is a new (dummy) constant. Nevertheless, the new TRS can be used now to recognize the first argument of `f` as redundant. That is, we are allowed to use the following rule `f = length(_:nil)` which completely avoids wasteful computations on redundant arguments. Hence, the different methods are complementary and an enhanced test might be developed by properly combine them.

10 Functional Logic Programming: Narrowing

Programs written in muti-paradigm functional-logic languages such as Curry (see e.g. those in Section 8) are usually not different from (equivalent) programs written in the (pure) functional language Haskell. The difference only shows up during the evaluation. In Curry, one can evaluate expressions containing logical variables (that are evaluated non-deterministically to deliver *computed answers* as in Prolog) while in Haskell only completely ground expressions can be (deterministically) evaluated to compute its *value*. In fact, Term Rewriting Systems are also used as abstract models of programs written in such languages, although narrowing, rather than rewriting, is usually the underlying computational mechanism (Hanus 1994).

Before the conclusions, let us discuss how the notions and techniques presented so far could be adapted to cope with more sophisticated, multi-paradigm functional-logic languages. The most popular operational principle to deal with logical variables within function calls is known as *narrowing*, as used in functional logic programming (see Hanus (1994) for a survey). Narrowing is an unification-based, parameter-passing mechanism which extends functional evaluation through goal solving capabilities as in logic programming. A *narrowing step* instantiates variables of an expression and then applies a reduction step to a redex of the instantiated expression. The instantiation of variables is usually computed by unifying a subterm of the entire expression with the left-hand side of some program equation. Narrowing provides completeness in the sense of logic programming, i.e., computation of answers, as well as functional programming, i.e., computation of normal forms. Formally, a term s narrows to t in $\mathscr{R}$, denoted by $s \leadsto_\sigma t$, iff there exists a non-variable position p of s,

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

a (standardized apart) rule $l \to r \in \mathscr{R}$, and a substitution σ such that $s|_p$ and l unify with mgu σ and $t = \sigma(s[r]_p)$.

Narrowing can be considered as a mapping (or semantics) $\mathsf{S} : \mathscr{T}(\mathscr{F},\mathscr{X}) \to \mathscr{P}(Subst(\mathscr{F},\mathscr{X}) \times \mathscr{T}(\mathscr{F},\mathscr{X}))$ that associates a set of pairs ⟨substitution,term⟩ to an input term (Hanus and Lucas 2001). The following is a typical evaluation semantics based on narrowing

$$\mathsf{eval}_{\mathsf{narr}}(t) = \{\langle \sigma, s\rangle \mid t \leadsto^*_\sigma s \wedge s \in \mathscr{T}(\mathscr{C},\mathscr{X})\}$$

The substitutions computed by narrowing are usually restricted to the variables of the input term. Within this semantic framework, the idea of redundancy for term rewriting as proposed in Definition 1 cannot be naïvely lifted to redundancy for narrowing (considering arbitrary input terms), as revealed by the following example.

Example 23

Consider the TRS of Example 1. The first argument of symbol `lastnew` is redundant w.r.t. $\mathsf{eval}_{\mathsf{narr}}$, i.e., for all contexts $C[\,]$ and for all $t, s \in \mathscr{T}(\mathscr{F},\mathscr{X})$ such that $root(t) = f$, $\mathsf{eval}_{\mathsf{narr}}(C[t]) = \mathsf{eval}_{\mathsf{narr}}(C[t[s]_i])$. For instance, with the input term $t =$ `lastnew(x,0:nil,s(0))`, we have $\mathsf{eval}_{\mathsf{narr}}(t[s]_1) = \{\langle id, \mathtt{s(0)}\rangle\}$ for all $s \in \mathscr{T}(\mathscr{F},\mathscr{X})$. This is because, in every lhs, the first argument is a variable that is never inspected in the corresponding rhs. However, the second argument of symbol `lastnew` is not redundant w.r.t. $\mathsf{eval}_{\mathsf{narr}}$. Consider the goal $t' =$ `lastnew(x,y,s(0))`, then $\mathsf{eval}_{\mathsf{narr}}(t') = \{\langle [\mathtt{y} \mapsto \mathtt{nil}], \mathtt{s(0)}\rangle, \langle [\mathtt{y} \mapsto \mathtt{w:nil}], \mathtt{s(0)}\rangle, \ldots\} \neq \{\langle id, \mathtt{s(0)}\rangle\} = \mathsf{eval}_{\mathsf{narr}}(t'[\mathtt{nil}]_2)$. The reason is that there exist many narrowing derivations for t':

$$\mathtt{lastnew(x,y,s(0))} \leadsto_{\{\mathtt{y}\mapsto\mathtt{nil}\}} \mathtt{s(0)},\ \mathtt{lastnew(x,y,s(0))} \leadsto^*_{\{\mathtt{y}\mapsto\mathtt{w:nil}\}} \mathtt{s(0)},\ \ldots$$

but only this one for $t'[\mathtt{nil}]_2$: $\mathtt{lastnew(x,nil,s(0))} \leadsto_{id} \mathtt{s(0)}$.

Thus, the general problem of analyzing redundancy w.r.t. the observable of computed answers is a challenging line of research that we pursue as future work (hence outside the scope of this paper). Nevertheless, we can still outline different possibilities for analyzing redundancy of arguments w.r.t. narrowing in some particular cases by applying the results in this paper.

Restriction to the variable case We have seen in Example 23 that the naïve notion of redundancy for narrowing is still fruitful when we consider the case of an argument in lhs's that always corresponds to a variable that is never inspected during the computation, i.e. the Variable Case in Section 6.1. For instance, using Theorem 10, we can identify that the first argument of symbol `lastnew` in Example 1 is redundant for narrowing and that also the second argument of symbol `loop` in Example 6 is redundant for narrowing.

Input terms with mode information Since the narrowing space is bigger than the rewriting space, the functional logic community (as well as the program transformation and partial evaluation community) usually restrict their interest to preserve the narrowing semantics $\mathsf{eval}_{\mathsf{narr}}$ for a fixed set of goals, similarly to the argument filtering technique of Leuschel and Sørensen (1996) for logic programming.

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

Example 24
Consider again the TRS of Example 1. Let us assume that we are only interested in the evaluation semantics of input terms that fit the shape `lastnew(`G`,`G`,`NG`)`, where G denotes a ground term and NG an arbitrary term. This is known as *mode information* in logic programming and implies that the first and second arguments of symbol `lastnew` are understood only as input data whereas the third argument is understood as input and output data. Then the techniques presented in Section 6 can be applied to the arguments that are labeled with G. For instance, the first and second arguments of `lastnew` will be detected as redundant for the mode `lastnew(`G`,`G`,`NG`)`.

As mentioned before, more research is needed in order to come upon a generally correct notion of redundancy w.r.t. narrowing, which leads to effective detection algorithms that pay off in practice. We believe that our results in this paper can be valuable for these studies.

11 Conclusion

This work provides the first results concerning the detection and removal of useless arguments in program functions. We developed our results in a stepwise manner. We have given a semantic definition of redundancy which takes the semantics S as a parameter. We have considered different (reduction) semantics, including the standard normalization semantics (typical of pure rewriting) and the evaluation semantics (closer to functional programming). We have provided some decidability results about redundancy of an argument and a first effective method for detecting redundancies, which is based on approximation techniques. We have also provided two more practical methods to recognize redundancy which allows us to simplify the general redundancy problem to the analysis of the rhs's of the program rules. All the three methods to detect redundancies are different and useful. Moreover, we think that all results in this paper are of independent interest and can be used for other applications in the fields of rule-based and multi-paradigm declarative programming.

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 some 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., Pettorossi and Proietti (1996a).

Furthermore, a prototype implementation of the (more practical) methods to detect redundancy together with the erasure procedure has been provided. The preliminary experiments performed with the prototype indicate that our approach is both practical and useful. We believe that the semantic grounds for redundancy

https://doi.org/10.1017/S1471068406002869 Published online by Cambridge University Press

analyses and elimination laid in this work may foster further insights and developments in the program optimization community and neighbouring fields.

Finally, apart from these comments, 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, $\mathsf{eval}_{\mathscr{R}}$-defined TRSs. We refer to Alpuente *et al.* (2002a) for details, where a comparison with approximation methods based on abstract interpretation can also be found.

Acknowledgements

We thank the anonymous referees for the useful remarks and suggestions which helped to improve the paper.

This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN 2004-7943-C04-02, the Generalitat Valenciana under grant GV03/25, and the ICT for EU-India Cross-Cultural Dissemination ALA/95/23/2003/077-054 project.

References

Aho, A., Sethi, R. and Ullman, J. 1986. *Compilers, Principles Techniques and Tools*. Addison-Wesley, Reading, MA.

Alpuente, M., Echahed, R., Escobar, S. and Lucas, S. 2002a. Redundancy of Arguments Reduced to Induction. In *Proc. of the 11th Int'l Workshop on Functional and (Constraint) Logic Programming WFLP'02*, M. Comini and M. Falaschi, Eds. Electronic Notes in Theoretical Computer Science, vol. 76. Elsevier Sciences Publisher, 100–200.

Alpuente, M., Escobar, S. and Lucas, S. 2002b. Removing Redundant Arguments of Functions. In *9th International Conference on Algebraic Methodology And Software Technology, AMAST 2002*, H. Kirchner and C. Ringeissen, Eds. Lecture Notes in Computer Science, vol. 2422. Springer-Verlag, Berlin, 117–131.

Alpuente, M., Escobar, S. and Lucas, S. 2006. Removing Redundant Arguments Automatically. CoRR cs.PL/0601039. Available at `http://arxiv.org/abs/cs.PL/0601039`.

Alpuente, M., Falaschi, M., Julián, P. and Vidal, G. 1997. Specialization of Lazy Functional Logic Programs. In *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*. ACM Sigplan Notices, vol. 32, number 12. ACM Press, New York, 151–162.

Alpuente, M., Falaschi, M. and Vidal, G. 1998. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems 20,* 4, 768–844.

Alpuente, M., Hanus, M., Lucas, S. and Vidal, G. 1999. Specialization of Inductively Sequential Functional Logic Programs. In *Proc. of the ACM SIGPLAN Conf. on Functional Programming, ICFP'99*, P. Lee, Ed. ACM Sigplan Notices, vol. 34, number 9. ACM Press, New York, 273–283.

Arts, T. and Giesl, J. 2001. A collection of examples for termination of term rewriting using dependency pairs. Tech. Rep. AIB-2001-09, RWTH Aachen, Germany.

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That*. Cambridge University Press.

Berardi, S., Coppo, M., Damiani, F. and Giannini, P. 2000. Type-based useless-code elimination for functional programs. In *Proceedings of SAIG 2000*. Lecture Notes in Computer Science, vol. 1924. Springer-Verlag, Berlin, 172–189.

Bert, D., Echahed, R. and Østvold, B. 1993. Abstract Rewriting. In *Proc. of Third Int'l Workshop on Static Analysis, WSA'93*. Lecture Notes in Computer Science, vol. 724. Springer-Verlag, Berlin, 178–192.

Burn, G. 1991. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Pitman, London.

Burn, G. L., Hankin, C. L. and Abramsky, S. 1986. The Theory of Strictness Analysis for Higher Order Functions. In *Programs as Data Objects*. Lecture Notes in Computer Science, vol. 217. Springer-Verlag, Berlin, 42–62.

Comon, H. 2000. Sequentiality, second order monadic logic, and tree automata. *Information and Computation 157*, 25–51.

Contejean E., Marché C., Monate B. and Urbain X. 2003. Proving termination of rewriting with C*i*ME. . In A. Rubio, editor,*Proc. of 6th International Workshop on Termination, WST'03*, pages 71-73, Technical Report DSIC II/15/03, Valencia, Spain, 2003. Available at `http://cime.lri.fr`.

Cousot, P. and Cousot, R. 1994. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proceedings of the 1994 International Conference on Computer Languages, ICCL'94*. IEEE Computer Society Press, Los Alamitos, California, Toulouse, France, 95–112.

Dauchet, M., Heuillard, T., Lescanne, P. and Tison, S. 1987. Decidability of the Confluence of Finite Ground Term Rewrite Systems and of Other Related Term Rewrite Systems. *Proc. of Second IEEE Symp. on Logic In Computer Science*, 353–359.

Dauchet, M., Heuillard, T., Lescanne, P. and Tison, S. 1990. Decidability of the Confluence of Ground Term Rewriting Systems. In *Information and Computation*. 88. Academic Press, New York, 187–201.

Durand, I. and Middeldorp, A. 1997. Decidable Call by Need Computations in Term Rewriting. In *Proc. of CADE'97*, W. McCune, Ed. Lecture Notes in Artificial Intelligence, vol. 1249. Springer-Verlag, Berlin, 4–18.

Gallagher, J. 1993. Tutorial on Specialisation of Logic Programs. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*. ACM, New York, 88–98.

Gallier, J. and Book, R. 1985. Reductions on tree replacement systems. *Theoretical Computer Science 37,* 2, 123–150.

Giesl J., Thiemann, R., Schneider-Kamp, P. and Falke, S. 2004. Automated Termination Proofs with AProVE. In V. van Oostrom, editor, *Proc. of 15h International Conference on Rewriting Techniques and Applications, RTA'04*, LNCS 3091:210-220, Springer-Verlag, Berlin, 2004. Available at `http://www-i2.informatik.rwth-aachen.de/AProVE`.

Glück, R. and Sørensen, M. 1994. Partial Deduction and Driving are Equivalent. In *Proc. of PLILP'94*. Lecture Notes in Computer Science, vol. 844. Springer-Verlag, Berlin, 165–181.

Godoy, G. and Tiwari, A. 2004. Deciding Fundamental Properties of Right-(Ground or Variable) Rewrite Systems by Rewrite Closure. In D. Basin and M. Rusinowitch, editors, *International Joint Conference on Automated Deduction, IJCAR'04*, LNAI 3097:91–106, Springer-Verlag, Berlin, 2004.

Gouranton, V. 1998. Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing. In *Proc. of the 5th International Static Analysis Symposium, SAS'98*. Lecture Notes in Computer Science, vol. 1503. Springer-Verlag, Berlin, 115–133.

Gramlich, B. 2001. On interreduction of semi-complete term rewriting systems. *Theoretical Computer Science 258,* 1–2, 435–451.

Hanus, M. 1994. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming 19&20*, 583–628.

Hanus, M., Antoy, S., Engelke, M., Höppner, K., Koj, J., Niederau, P., Sadre, R. and Steiner, F. 2003a. PAKCS 1.5.0: The Portland Aachen Kiel Curry System User Manual. Tech. rep., University of Kiel, Germany.

Hanus, M., Antoy, S., Kuchen, H., López-Fraguas, F., Lux, W., Moreno Navarro, J., and Steiner, F. 2003b. Curry: An Integrated Functional Logic Language (version 0.8). Available at `http://www.informatik.uni-kiel.de/ curry`.

M. Hanus and S. Lucas. 2001. An Evaluation Semantics for Narrowing-Based Functional Logic Languages. *Journal of Functional and Logic Programming*, 2001(2):1–43.

Hendrix J., Clavel M. and Meseguer J. 2005. A Sufficient Completeness Reasoning Tool for Partial Specifications. In Jürgen Giesl, editor, *Proc. of 16h International Conference on Rewriting Techniques and Applications, RTA'05*, LNCS 3467:165–174, Springer-Verlag, Berlin, 2005. Available at `http://maude.cs.uiuc.edu/tools/scc`.

Hofbauer, D. 2003. An upper bound on the derivational complexity of Knuth-Bendix orderings. *Information and Computation* 183(1):43–56.

Huet, G. and Lévy, J.-J. 1991. Computations in Orthogonal Term Rewriting Systems, Part I + II. In *Computational logic: Essays in honour of J. Alan Robinson*. The MIT Press, Cambridge, MA, 395–414 and 415–443.

Hughes, J. 1988. Backwards Analysis of Functional Programs. In *IFIP Workshop on Partial Evaluation and Mixed Computation* (Amsterdam), D. Bjørner, A. Ershov, and N. Jones, Eds. 187–208.

Jacquemard, F. 1996. Decidable approximations to term rewriting systems. In *Proc. of 7th International Conference on Rewriting Techniques and Applications, RTA'96*, H. Ganzinger, Ed. Lecture Notes in Computer Science, vol. 1103. Springer-Verlag, Berlin, 362–376.

Jensen, T. P. 1991. Strictness Analysis in Logical Form. In *Proc of International Conference on Functional Programming Languages and Computer Architecture*, R. J. M. Hughes, Ed. Lecture Notes in Computer Science, vol. 523. Springer-Verlag, Berlin, 352–366.

Jones, S. B. and Métayer, D. L. 1989. Compile-time garbage collection by sharing analysis. In *Proc of International Conference on Functional Programming Languages and Computer Architecture*. ACM Press, New York, 54–74.

Kapur, D., Narendran, P. and Zhang, Z. 1987. On sufficient-completeness and related properties of term rewriting systems. *Acta Informatica 24*, 395–416.

Kapur, D., Narendran, P., Rosenkrantz, D.J. and Zhang, Z. 1991. Sufficient-completeness, ground-reducibility, and their complexity. *Acta Informatica 28*, 311–350.

Kennaway, R., van Oostrom, V. and de Vries, F. 1996. Meaningless terms in rewriting. In *Proceedings of the ALP'96*. Lecture Notes in Artificial Intelligence, vol. 1139. Springer-Verlag, 254–268.

Klop, J. 1992. Term Rewriting Systems. In *Handbook of Logic in Computer Science*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Vol. 3. Oxford University Press, Oxford, 1–116.

Knoop, J., Rüthing, O. and Steffen, B. 1994. Partial Dead Code Elimination. *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI'94) 29,* 6, 147–158.

Kobayashi, N. 2000. Type-based useless variable elimination. In *Proceedings of PEPM-00*. ACM Press, New York, 84–93.

Kounalis, E. 1985. Completeness in data type specifications. In *Proc. of European Conference on Computer Algebra, EUROCAL'85*, B. Caviness, Ed. Lecture Notes in Computer Science, vol. 204. Springer-Verlag, Berlin, 348–362.

Kuper, J. 1994. Partiality in logic and computation. aspects of undefinedness. Ph.D. thesis, Universiteit Twente.

Leuschel, M. 1998. On the Power of Homeomorphic Embedding for Online Termination. In *Proc. of the 5th International Static Analysis Symposium, SAS'98*, G. Levi, Ed. Lecture Notes in Computer Science, vol. 1503. Springer-Verlag, Berlin, 230–245.

Leuschel, M. and Martens, B. 1995. Partial Deduction of the Ground Representation and Its Application to Integrity Checking. Tech. Rep. CW 210DSIC-II/8/02, K.U. Leuven.

Leuschel, M. and Sørensen, M. H. 1996. Redundant Argument Filtering of Logic Programs. In *Proceedings of the 6th International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, J. Gallager, Ed. Lecture Notes in Computer Science, vol. 1207. Springer-Verlag, Berlin, Stockholm, Sweden, 83–103.

Liu, Y. A. and Stoller, S. D. 2002. Eliminating dead code on recursive data. *Science of Computer Programming*.

Lucas, S. 2001. Transfinite Rewriting Semantics for Term Rewriting Systems. In *Proc. of 12th Int'l Conf. on Rewriting Techniques and Applications, RTA'01*, A. Middeldorp, Ed. Lecture Notes in Computer Science, vol. 2051. Springer-Verlag, Berlin, 216–230.

Mitsuhashi, I. and Oyamaguchi, M. and Ohta, Y. and Yamada, T. 2004. The Joinability and Unification Problems for Confluent Semi-constructor TRSs. In *Proc. of 15th Int'l Conf. Rewriting Techniques and Applications, RTA'04*, V. van Oostrom, Ed. Lecture Notes in Computer Science, vol. 3091. Springer-Verlag, Berlin, 285–300.

Mycroft, A. 1980. The theory and practice of transforming call by need into call by value. In *4th International Symposium on Programming*, B. Robinet, Ed. Lecture Notes in Computer Science, vol. 83. Springer-Verlag, Berlin, 269–281.

Mycroft, A. and Norman, A. 1992. Optimising compilation. Part ii: lazy functional languages. In *XIX Seminar on Current Trends in Theory and Practice of Informatics, SOFSEM'92*, Ždiar, Czechoslovakia. Available at `http://www.cl.cam.ac.uk/~am/papers/sofsem92b.ps.gz`.

Oyamaguchi, M. 1986. The reachability problems for quasi-ground for term rewriting systems. *Journal of Information Processing 9,* 4, 232–236.

Oyamaguchi, M. 1990. The reachability and joinability problems for right-ground term rewriting systems. *Journal of Information Processing 13,* 3, 347–354.

Padawitz, P. 1988. *Computing in Horn Clause Theories*. EATCS Monographs on Theoretical Computer Science, vol. 16. Springer-Verlag, Berlin.

Park, Y. G. and Goldberg, B. 1992. Escape Analysis on Lists. *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI'92) 27,* 7, 116–127.

Pettorossi, A. and Proietti, M. 1994. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming 19, 20*, 261–320.

Pettorossi, A. and Proietti, M. 1996a. A Comparative Revisitation of Some Program Transformation Techniques. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation.* Lecture Notes in Computer Science, vol. 1110. Springer-Verlag, Berlin, 355–385.

Pettorossi, A. and Proietti, M. 1996b. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys 28,* 2, 360–414.

Plasmeijer, R. and van Eekelen, M. 1993. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley.

Rabin, M. O. 1969. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society 141*, 1–35.

Reps, T. and Turnidge, T. 1996. Program Specialization via Program Slicing. In *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, O. Danvy, R. Glück, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1110. Springer-Verlag, Berlin, 409–429.

Schoenig, S. and Ducasse, M. 1996. A Backward Slicing Algorithm for Prolog. In *Proc. of the 3rd International Static Analysis Symposium, SAS'96*. Lecture Notes in Computer Science, vol. 1145. Springer-Verlag, Berlin, 317–331.

Sekar, R., Pawagi, S. and Ramakrishnan, I. 1990. Small domains spell fast strictness analysis. In *16th Annual ACM Symposium on Principles of Programming Languages, POPL'89*. ACM Press, New York, 169–183.

Szilagyi, G., Gyimothy, T., and Maluszynski, J. 2002. Static and Dynamic Slicing of Constraint Logic Programs. *Journal of Automated Software Engineering 9,* 1, 41–65.

TeReSe, Ed. 2003. *Term Rewriting Systems*. Cambridge University Press, Cambridge.

Thatcher, J. W. and Wright, J. B. 1968. Generalized finite automata with an application to a decision problem of second-order logic. *Math. Systems Theory 2*, 57–82.

Thomas, W. 1990. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, Mass, 133–191.

Tip, F. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages 3*, 121–189.

Verma, M.-R. 2002, Algorithms and Reductions for Rewriting Problems II. *Information Processing Letters* 84(4):227–233.

Wadler, P. and Hughes, R. 1987. Projections for Strictness Analysis. In *Proc of International Conference on Functional Programming Languages and Computer Architecture*. Lecture Notes in Computer Science, vol. 274. Springer-Verlag, Berlin, 385–407.

Wand, M. and Siveroni, I. 1999. Constraint systems for useless variable elimination. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'99)*. ACM Press, New York, 291–302.

Weiser, M. 1984. Program Slicing. *IEEE Transactions on Software Engineering 10,* 4, 352–357.