

# Termination and complexity bounds for SAFE programs\*

Salvador Lucas

Dep. de Sistemas Informáticos y Computación  
Universidad Politécnica de Valencia  
slucas@dsic.upv.es

Ricardo Peña

Dep. de Sistemas Informáticos y Computación  
Universidad Complutense de Madrid  
ricardo@sip.ucm.es

## Abstract

Safe is a first-order eager functional language with facilities for programmer-controlled destruction and copying of data structures and is intended for compile-time analysis of memory consumption. In Safe, heap memory consumption depends on the length of recursive calls chains. Ensuring termination of Safe programs (or of particular function calls) is therefore essential to implement these features. Furthermore, being able to giving bounds to the chain length required by such terminating calls becomes essential in computing space bounds.

In this paper, we investigate how to analyze termination of Safe programs by using standard term rewriting techniques, i.e., by transforming Safe programs into term rewriting systems whose termination can be automatically analyzed by means of existing tools. Furthermore, we investigate how to use proofs of termination which combine the dependency pairs approach together with polynomial interpretations to obtain suitable bounds to the length of chains of recursive calls in Safe programs.

## 1 Introduction

Safe [20, 21, 18] is a first-order eager functional language with facilities for programmer-controlled destruction and copying of data structures and is intended for compile-time

analysis of memory consumption. In Safe, the allocation and deallocation of compiler-defined memory regions for data structures is associated to the function application. So, heap memory consumption depends on the length of recursive calls chains.

In order to compute space bounds for such chains it is essential first to compute bounds to their lengths and, in turn, it is previously essential to ensure termination of such functions.

Both termination and complexity bounds of programs have been investigated in the abstract framework of Term Rewriting Systems (TRSs [2, 19]). In this paper we investigate how to use rewriting techniques for proving termination of Safe programs and giving appropriate bounds to the number of recursive calls of Safe programs as a first step to compute space bounds.

A suitable way to prove termination of programs written in declarative programming languages like Haskell or Maude [7] is translating them into (variants of) term rewriting systems and then using techniques and tools for proving termination of rewriting. See [10, 12] for recent proposals of concrete procedures and tools which apply to the aforementioned programming languages. In this paper we introduce a transformation for proving termination of Safe programs by translating them into Term Rewriting Systems.

Polynomial interpretations have been extensively investigated as suitable tools to address different issues in term rewriting [2]. For instance, the limits of polynomial interpretations regarding their ability to prove termination of rewrite systems were first investigated in [14] by considering the *derivational*

---

\*Work partially supported by the EU (FEDER) and the Spanish MEC, under grant TIN 2004-7943-C04. Salvador Lucas was partially supported by the EU (FEDER) and the Spanish MEC grant HA 2006-0007, and by the Generalitat Valenciana under grant GV06/285. Ricardo Peña was partially supported by the Madrid Region Government under grant S-0505/TIC/0407 (PROMESAS).

*complexity* of polynomially terminating TRSs, i.e., the upper bound of the lengths of arbitrary (but finite) derivations issued from a given term (of size  $n$ ) in a terminating TRS. Hofbauer has shown that the derivational complexity of a terminating TRS can be better approximated if polynomial interpretations over the reals (instead of the more traditional polynomial interpretations over the naturals) are used to prove termination of the TRS [13].

Complexity analysis of first order functional programs (or TRSs) has also been successfully addressed by using polynomial interpretations [3, 4, 5, 6]. The aim of these papers is to classify TRSs in different (TIME or SPACE) complexity classes according to the (least) kind of polynomial interpretation which is (weakly) compatible with the TRS. Recent approaches combine the use of *path orderings* [9] to ensure both termination together with suitable polynomial interpretations for giving bounds to the length of the rewrite sequences (which are known finite due to the termination proof), see [5]. Polynomials which are used in this setting are *weakly monotone*, i.e., if  $x \geq y$  then  $P(\dots, x, \dots) \geq P(\dots, y, \dots)$ . This is in contrast with the use of polynomials in proofs of polynomial termination [15], where *monotony* is required (i.e., whenever  $x > y$ , we have  $P(\dots, x, \dots) > P(\dots, y, \dots)$ ). However, when using polynomials in proofs of termination using the dependency pair approach [1], monotony is not longer necessary and we can use weakly monotone polynomials again [8, 16]. The real advantage is that, we can now avoid the use of path orderings to ensure termination: with the same polynomial interpretation we can both prove termination and as we show in this paper, obtain suitable complexity bounds. Furthermore, since the limits of using path orderings to prove termination of rewrite systems are well-known, and they obviously restrict the variety of programs they can deal with, we are able to improve on the current techniques.

## 2 Preliminaries

A binary relation  $R$  on a set  $A$  is *terminating* (or well-founded) if there is no infinite sequence  $a_1 R a_2 R a_3 \dots$ . Given

$f : A^k \rightarrow A$  and  $i \in \{1, \dots, k\}$ , we say that  $R$  is monotonic on the  $i$ -th argument of  $f$  (or that  $f$  is  $i$ -monotone regarding  $R$ ) if  $f(x_1, \dots, x, \dots, x_k) R f(x_1, \dots, y, \dots, x_k)$  whenever  $x R y$ , for all  $x, y, x_1, \dots, x_k \in A$ . We say that  $R$  is *monotonic* regarding  $f$  (or that  $f$  is  $R$ -monotone) if  $R$  is  $i$ -monotonic on the  $i$ -th argument of  $f$  for all  $i$ ,  $1 \leq i \leq k$ . A transitive and reflexive relation  $\succeq$  on  $A$  is a quasi-ordering. A transitive and irreflexive relation  $>$  on  $A$  is an ordering.

**Terms and term rewriting** Throughout the paper,  $\mathcal{X}$  denotes a countable set of variables and  $\mathcal{F}$  denotes a signature, i.e., a set of function symbols  $\{f, g, \dots\}$ , each having a fixed arity given by a mapping  $ar : \mathcal{F} \rightarrow \mathbb{N}$ . The set of terms built from  $\mathcal{F}$  and  $\mathcal{X}$  is  $\mathcal{T}(\mathcal{F}, \mathcal{X})$ . A *context* is a term  $C[\ ]$  with a ‘hole’ (formally, a fresh constant symbol). A rewrite rule is an ordered pair  $(l, r)$ , written  $l \rightarrow r$ , with  $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ ,  $l \notin \mathcal{X}$  and  $Var(r) \subseteq Var(l)$ . A TRS is a pair  $\mathcal{R} = (\mathcal{F}, R)$  where  $R$  is a set of rewrite rules. Given  $\mathcal{R} = (\mathcal{F}, R)$ , we consider  $\mathcal{F}$  as the disjoint union  $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$  of symbols  $c \in \mathcal{C}$ , called *constructors* and symbols  $f \in \mathcal{D}$ , called *defined functions*, where  $\mathcal{D} = \{root(l) \mid l \rightarrow r \in R\}$  and  $\mathcal{C} = \mathcal{F} - \mathcal{D}$ . Given a TRS  $\mathcal{R}$ , a term  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  rewrites to  $s$ , written  $t \rightarrow_{\mathcal{R}} s$ , if  $t = C[\sigma(l)]$  and  $s = C[\sigma(r)]$ , for some rule  $\rho : l \rightarrow r \in R$ , context  $C[\ ]$  and substitution  $\sigma$ . A TRS  $\mathcal{R}$  is terminating if  $\rightarrow_{\mathcal{R}}$  is terminating.

**Term orderings and algebraic interpretations** Term orderings can be obtained by giving appropriate *interpretations* to the function symbols of a signature. Given a signature  $\mathcal{F}$ , an  $\mathcal{F}$ -algebra is a pair  $\mathcal{A} = (A, \mathcal{F}_A)$ , where  $A$  is a set and  $\mathcal{F}_A$  is a set of mappings  $f_A : A^k \rightarrow A$  for each  $f \in \mathcal{F}$  where  $k = ar(f)$ . For a given valuation mapping  $\alpha : \mathcal{X} \rightarrow A$ , the evaluation mapping  $[\alpha] : \mathcal{T}(\mathcal{F}, \mathcal{X}) \rightarrow A$  is inductively defined by  $[\alpha](x) = \alpha(x)$  if  $x \in \mathcal{X}$  and  $[\alpha](f(t_1, \dots, t_k)) = f_A([\alpha](t_1), \dots, [\alpha](t_k))$  for  $x \in \mathcal{X}$ ,  $f \in \mathcal{F}$ ,  $t_1, \dots, t_k \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ . Given a term  $t$  with  $Var(t) = \{x_1, \dots, x_n\}$ , we write  $[t]$  to denote the *function*  $F_t : A^n \rightarrow A$  given by  $F_t(a_1, \dots, a_n) = [\alpha_{(a_1, \dots, a_n)}](t)$  for each tuple

$(a_1, \dots, a_n) \in A^n$ , where  $\alpha_{(a_1, \dots, a_n)}(x_i) = a_i$  for  $1 \leq i \leq n$ .

A *quasi-ordered*  $\mathcal{F}$ -algebra, is a triple  $(A, \mathcal{F}_A, \succeq_A)$ , where  $(A, \mathcal{F}_A)$  is an  $\mathcal{F}$ -algebra and  $\succeq_A$  is a quasi-ordering on  $A$ . Then, we can define a stable quasi-ordering  $\succeq$  on terms given by  $t \succeq s$  if and only if  $[\alpha](t) \succeq_A [\alpha](s)$ , for all  $\alpha : \mathcal{X} \rightarrow A$ .

### 3 The SAFE language

Safe was first introduced in [20] as a research platform to investigate analyses related to sharing of data structures and to memory consumption. Currently it is provided with a type system guaranteeing that all well-typed programs will be free of dangling pointers at runtime, in spite of the memory destruction facilities provided by the language. More information can be found in [21] and [18].

There are two versions of Safe: full-Safe, in which programmers are supposed to write their programs, and Core-Safe (the compiler transformed version of full-Safe), in which all program analyses are defined.

Full-Safe syntax is close to Haskell's. The main difference is that Safe is first-order at this moment. Safe admits two basic types (*booleans* and *integers*), algebraic datatypes (which are introduced by means of the usual **data** declarations), and the definition of functions by means of conditional equations with the usual facilities for pattern matching, use of **let** and **case** expressions, and **where** clauses. No recursion is possible inside **let** expressions and **where** clauses and no local function definition can be given.

A Safe program consists of a sequence of (possibly recursive) function definitions together with a *main expression*. Let  $\overline{x_i}^n$  denote the sequence  $x_1, \dots, x_n$  in the following:

$$\begin{aligned} prog &\rightarrow f_1 \overline{x_{1j}}^{n_1} = e_1; \\ &\dots \\ &f_r \overline{x_{rj}}^{n_r} = e_r; \\ &e \end{aligned}$$

where the only free variables in  $e_i, 1 \leq i \leq r$  are  $\overline{x_{ij}}^{n_i}$  and  $f_1, \dots, f_i$ , and the only free variables in  $e$  are  $f_1, \dots, f_r$ .

Additionally, we can specify a *destructive* pattern matching operation by using symbol

! after the pattern. The intended meaning of this operator is the destruction of the cell which is associated to the constructor symbol thus allowing its reuse later.

The following merge-sort program uses a constant heap space to implement the sorting of the list.

```

splitD :: ∀a, ρ. Int → [a]!@ρ → ρ → ([a]!@ρ, [a]!@ρ)@ρ
splitD 0 xs! = ([], xs!)
splitD n []! = ([], [])
splitD n (x : xs)! = (x : xs1, xs2)
  where (xs1, xs2) = splitD (n - 1) xs

mergeD :: ∀a, ρ. [a]!@ρ → [a]!@ρ → ρ → [a]!@ρ
mergeD []! ys! = ys!
mergeD xs! []! = xs!
mergeD (x : xs)! (y : ys)!
  | x ≤ y = x : mergeD xs (y : ys!)
  | otherwise = y : mergeD (x : xs!) ys

msortD :: ∀a, ρ. [a]!@ρ → ρ → [a]!@ρ
msortD xs
  | n ≤ 1 = xs!
  | otherwise = mergeD (msortD xs1) (msortD xs2)
  where (xs1, xs2) = splitD (n `div` 2) xs
        n = length xs

```

This is a consequence of the destructive constant-space versions *splitD* and *mergeD* of the functions which respectively split a list into two pieces and merge two sorted lists. Types which are shown in the program above are inferred by the compiler. A symbol ! in a type signature indicates that the corresponding data structure is destroyed by the function. Variables  $\rho$  are polymorphic and indicate the region where the data structure 'lives'.

#### 3.1 Core-Safe syntax

The Safe compiler first performs a *region inference* which determines which region have to be used for each construction. A function has two associated memory regions: a *working* region which can be addressed by using the reserved identifier *self* and an *output* region which is passed as a parameter. For this reason, the low-level syntax, called Core-Safe requires an additional parameter  $r$  both in some function calls and in expressions such as  $(C \overline{x_i}^n)@r$  which denotes a construction, and  $x@r$  which denotes the *copy* of the structure with root labeled  $x$  into a region  $r$ . The compiler also *flattens* the expressions in such a way that the application of functions is made on constants or variables only. Also, **where** clauses are translated into **let** expressions, and boolean conditions in the guards are translated into **case**

expressions. Bound variables are also conveniently renamed to avoid name clashes.

The syntax of Core-Safe is shown in Figure 1. Note that constructions can only occur on *binding expressions*  $be$  inside **let** expressions. The normal form of an expression is either a basic constant  $c$ , or a pointer  $p$  to a construction. We assume the existence of a heap which keeps track of the correspondence between pointers and constructions. The complete operational semantics can be found in [21].

Function *splitD* defined in the Safe program above is translated into the following Core-Safe definition:

```
splitD n xs r =
  case n of
    0 -> ([@r, xs!}@r
  _ -> case! xs of
    [] -> ([@r, []@r}@r
      : x xx -> let z = let n' = n-1 in
                  splitD n' xx @ r in
                let xs1 = case z of
                    (ys1, ys2) -> ys1 in
                let xs2 = case z of
                    (zs1, zs2) -> zs2 in
                let xs1' = (: x xs1}@r in
                    (xs1', xs2}@r
```

#### 4 Transformation from Core-SAFE to CTRS

In this section we describe a transformation of Core-SAFE programs into *Conditional Term Rewriting Systems (CTRSs)*. For the purpose of the transformation, we can even simplify the Core-SAFE syntax, because information concerning destructive patterns and regions is not relevant for termination purposes. Thus, we use the simplified syntax in Figure 2. We assume that each **case** expression in a function definition has a unique integer label  $k$ . The transformation is defined by means of the following auxiliary functions:

1.  $trP$  which takes a sequence of Core-Safe function definitions and returns a CTRS.
2.  $trF$  which takes a function definition and returns a set of conditional rewrite rules.
3.  $trR$  which given an expression  $e$ , or a binding expression  $be$ , the set  $V$  of its free variables, and a condition  $C = s_1 \rightarrow t_1, \dots, s_k \rightarrow t_k$  consisting of atomic

(rewrite) conditions  $s_i \rightarrow t_i$ , returns the right-hand side of a rule together with its conditional part. In fact, we treat  $C$  as a list; if the conditional part  $C = []$  then the generated rule has no conditional part.

4.  $trL$  which, given a expression  $e$  and the set  $V$  of its free variables yields a left part of a condition, and a sequence of atomic conditions to its left.

Assume that  $var(V)$  assigns the variables in  $V$  to a given term  $t$  in a fixed ordering. The transformation is given as follows:

$$trP(\overline{def}^n) \stackrel{\text{def}}{=} \bigcup_{i=1}^n trF(def_i)$$

$$trF(f \overline{x}^n = e) \stackrel{\text{def}}{=} f(x_1, \dots, x_n) \rightarrow trR(e, fv(e), [])$$

$$trR(c, V, C) \stackrel{\text{def}}{=} c \Leftarrow C$$

$$trR(x, V, C) \stackrel{\text{def}}{=} x \Leftarrow C$$

$$trR(C_r \overline{a}^n, V, C) \stackrel{\text{def}}{=} C_r(a_1, \dots, a_n) \Leftarrow C$$

$$trR(f \overline{a}^n, V, C) \stackrel{\text{def}}{=} f(a_1, \dots, a_n) \Leftarrow C$$

$$trR(k : \text{case } x \text{ of } \overline{C_i \overline{x}_{ij}^{n_i}} \rightarrow e_i^n, V, C) \stackrel{\text{def}}{=} \{ \text{case}_k(x, var(V)) \Leftarrow C \} \cup \{ \text{case}_k(C_i(x_{i1}, \dots, x_{in_i}), var(V)) \rightarrow trR(e_i, fv(e_i), []) \mid i \in \{1..n\} \}$$

$$trR(\text{let } x_1 = e_1 \text{ in } e_2, V, C) \stackrel{\text{def}}{=} trR(e_2, fv(e_2), C \text{ ++ } [trL(e_1, fv(e_1)) \rightarrow x_1])$$

$$trL(e, V) \stackrel{\text{def}}{=} trR(e, V, []) \text{ if } e \in \{c, x, C_r \overline{a}^n, f \overline{a}^n, \text{case}\}$$

$$trL(\text{let } x_1 = e_1 \text{ in } e_2, V) \stackrel{\text{def}}{=} [trL(e_1, fv(e_1)) \rightarrow x_1, ] \text{ ++ } trL(e_2, fv(e_2))$$

Our running example would be transformed into the following CTRS:

```
splitD(n, xs) -> case1(n, n, xs)
case1(0, n, xs) -> Tup( Nil, xs)
case1(S(x), n, xs) -> case2(xs, n)
case2( Nil, n) -> Tup( Nil, Nil)
case2( Cons(x, xx), n) -> Tup( xs1', xs2)
    <= n-1 -> n', splitD(n', xx) -> z,
    case3(z) -> xs1, case4(z) -> xs2,
    Cons(x, xs1) -> xs1'
case3(Tup(ys1, ys2)) -> ys1
case4(Tup(zs1, zs2)) -> zs2
```

**Proposition 1** *Every Core-SAFE program  $\mathcal{P}$  is transformed into a left-linear, non-overlapping deterministic 3-CTRS  $trP(\mathcal{P})$  which is, therefore, confluent.*

**Proposition 2** *Given a Core-SAFE program  $\mathcal{P}$  and its transformed 3-CTRS  $\mathcal{R} = trP(\mathcal{P})$  the main expression  $e$  of  $\mathcal{P}$  terminates according to Safe semantics if and only if the term  $t_e$  associated to  $e$  terminates in  $\mathcal{R}$ . Furthermore, in every term (except the last one, if it*

$prog$	$\rightarrow$	$dec_1; \dots; dec_n; e$	
$dec$	$\rightarrow$	$f \overline{x_i^n} r = e$	{single-recursive, polymorphic function}
		$  f \overline{x_i^n} = e$	{This does not return a new data structure}
$a$	$\rightarrow$	$c$	{atom is a basic constant}
		$  x$	{or a variable}
$e$	$\rightarrow$	$a$	
		$  x @r$	{copy}
		$  x!$	{reuse}
		$  (f \overline{a_i^n}) @r$	{function application}
		$  (f \overline{a_i^n})$	
		$  \mathbf{let} x_1 = be \mathbf{in} e$	{non-recursive, monomorphic}
		$  \mathbf{case} x \mathbf{of} \overline{alt_i^n}$	{read-only case}
		$  \mathbf{case!} x \mathbf{of} \overline{alt_i^n}$	{destructive case}
$alt$	$\rightarrow$	$C \overline{x_i^n} \rightarrow e$	{algebraic datatype alternative}
		$  \_ \rightarrow e$	{default alternative for literal <b>case</b> }
$be$	$\rightarrow$	$C \overline{a_i^n} @r$	{constructor application}
		$  e$	

Figure 1: Syntax of Core-Safe

$prog$	$\rightarrow$	$dec_1; \dots; dec_n; e$	
$dec$	$\rightarrow$	$f \overline{x_i^n} = e$	{A single version of function declaration}
$a$	$\rightarrow$	$c$	{basic constant}
		$  x$	{variable (replaces reuse and copy expressions)}
$e$	$\rightarrow$	$a$	
		$  f \overline{a_i^n}$	{A single version of function application}
		$  \mathbf{let} x_1 = be \mathbf{in} e$	{non-recursive, monomorphic}
		$  \mathbf{case} x \mathbf{of} \overline{alt_i^n}$	{a single version of case}
$alt$	$\rightarrow$	$C \overline{x_i^n} \rightarrow e$	
		$  \_ \rightarrow e$	
$be$	$\rightarrow$	$C \overline{a_i^n}$	{constructor application (region is irrelevant)}
		$  e$	

Figure 2: Simplified Core-SAFE

exists) of the reduction sequence of  $t_e$  there is only one redex.

We remind that a 3-CTRS satisfies  $var(r) \subseteq var(l) \cup var(C)$  for every conditional rule of the form  $l \rightarrow r \Leftarrow C$ . The deterministic property means that the variables of the righthand side of every condition  $s_i \rightarrow t_i$  of  $C$  are introduced before they are used in the lefthand side of a subsequent condition  $s_j \rightarrow t_j$ .

Now we can apply standard transformations from deterministic 3-CTRS into unconditional TRSs [19, Def. 7.2.48]. If  $\mathcal{R}$  is a 3-CTRS, let us call  $U(\mathcal{R})$  to the TRS resulting from the transformation. For instance, in our running example  $U(\mathcal{R})$  would be the following TRS:

```

splitD(n,xs) -> case1(n,n,xs)
case1(0,n,xs) -> Tup(Nil,xs)
case1(S(x),n,xs) -> case2(xs,n)
case2(Nil,n) -> Tup(Nil,Nil)

```

```

case2(Cons(x,xx),n) -> U1(n-1,x,xx)
U1(n',x,xx) -> U2(splitD(n',xx),x)
U2(z,x) -> U3(case3(z),z,x)
U3(xs1,z,x) -> U4(case4(z),x,xs1)
U4(xs2,x,xs1) -> U5(Cons(x,xs1),xs2)
U5(xs1',xs2) -> Tup(xs1',xs2)
case3(Tup(ys1,ys2)) -> ys1
case4(Tup(zs1,zs2)) -> zs2

```

**Proposition 3** *For every Core-SAFE program  $\mathcal{P}$ , the TRS  $U(trP(\mathcal{P}))$  satisfies the following properties:*

1. It consists of non-overlapping rules. Moreover, all the lefthand sides are of the form  $f(p_1, \dots, p_n)$  where the  $p_i$  are flat patterns.
2. The righthand sides have at most a nesting depth of 1. In the worst case they are of the form  $g(e_1, \dots, e_n)$  with  $g$  being a function symbol and all the  $e_i$  being

either variables, flat patterns, or terms  $f(a_1, \dots, a_m)$  with  $f$  being a function symbol and the  $a_j$  variables or basic constants.

It is a standard result [19, Prop. 7.2.50] that the termination of  $U(\mathcal{R})$  implies the termination of  $\mathcal{R}$ .

## 5 Termination and complexity bounds

Termination of rewriting can be proved by using the dependency pairs approach [1]. Given a TRS  $\mathcal{R} = (\mathcal{C} \uplus \mathcal{D}, R)$  the set  $\text{DP}(\mathcal{R})$  of *dependency pairs* for  $\mathcal{R}$  is given as follows: if  $f(t_1, \dots, t_m) \rightarrow r \in R$  and  $r = C[g(s_1, \dots, s_n)]$  for some defined symbol  $g \in \mathcal{D}$ , and context  $C[\cdot]$ , and  $s_1, \dots, s_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ , then  $f^\sharp(t_1, \dots, t_m) \rightarrow g^\sharp(s_1, \dots, s_n) \in \text{DP}(\mathcal{R})$ , where  $f^\sharp$  and  $g^\sharp$  are new fresh symbols associated to  $f$  and  $g$  respectively. Termination of rewriting can be ensured by inspecting the *cycles* of the *dependency graph* associated to the TRS  $\mathcal{R}$ . The nodes of the dependency graph are the dependency pairs in  $\text{DP}(\mathcal{R})$ ; we refer the reader to [1] for details about how to build it.

An argument filtering  $\pi$  for a signature  $\mathcal{F}$  is a mapping that assigns to every  $k$ -ary function symbol  $f \in \mathcal{F}$  an argument position  $i \in \{1, \dots, k\}$  or a (possibly empty) list  $[i_1, \dots, i_m]$  of argument positions with  $1 \leq i_1 < \dots < i_m \leq k$ . Argument filterings apply to terms to remove appropriate symbols and subterms (see [1]). Furthermore, when  $S$  is a subset of rules, we write  $\pi(S)$  to denote the set  $\{\pi(s) \rightarrow \pi(t) \mid s \rightarrow t \in S\}$ .

A reduction pair  $(\succeq, \sqsupset)$  consists of a stable and weakly monotonic quasi-ordering  $\succeq$ , and a stable and well-founded ordering  $\sqsupset$  satisfying either  $\succeq \circ \sqsupset \subseteq \sqsupset$  or  $\sqsupset \circ \succeq \subseteq \sqsupset$ . The following results justify the use of reduction pairs in proofs of termination using dependency pairs.

**Theorem 1 (DP termination [1])** *A TRS  $\mathcal{R}$  is terminating if and only if there is a reduction pair  $(\succeq, \sqsupset)$  such that  $\mathcal{R} \subseteq \succeq$  and  $\text{DP}(\mathcal{R}) \subseteq \sqsupset$ .*

Let us assume that we use Theorem 1 and—by hand or by using a suitable tool—we find

a polynomial interpretation of the TRS resulting from transforming a Core-Safe program. Let us call  $\llbracket f^\sharp \rrbracket$  to the polynomial interpreting the symbol  $f^\sharp$  associated to the Core-Safe function symbol  $f$ . This polynomial is guaranteed to remain non-negative for non-negative arguments and to decrease at each dependency pair. Moreover, in contrast to the polynomials obtained by using Theorem 2 (see below), there is a single polynomial interpreting each symbol  $f^\sharp$ .

**Proposition 4 (polynomial bounds)** *If  $\llbracket f^\sharp \rrbracket$  is the polynomial (satisfying Theorem 1 and) interpreting the symbol  $f^\sharp$  associated to a  $n$ -ary function symbol  $f$  in a Core-SAFE program and  $x_1, \dots, x_n$  are interpreted as the sizes of the input arguments to  $f$ , then the number  $N(x_1, \dots, x_n)$  of recursive calls to  $f$  with arguments  $t_1, \dots, t_n$  of sizes  $x_1, \dots, x_n$ , respectively, is bounded by  $\llbracket f^\sharp \rrbracket(x_1, \dots, x_n)$ , i.e.  $N(x_1, \dots, x_n) \leq \llbracket f^\sharp \rrbracket(x_1, \dots, x_n)$*

Consider the TRS  $U(\mathcal{R})$  obtained in Section 4 for our running example. The following polynomial interpretation:

```
[pred](X) = 1/2.X
[S](X) = 2.X + 1
[splitD](X1,X2) = 0
[case1](X1,X2,X3) = 0
[0] = 0
[tup](X1,X2) = 0
[nil] = 0
[case2](X1,X2) = 0
[cons](X1,X2) = X2 + 2
[U1](X1,X2,X3) = 0
[U2](X1,X2) = 0
[case3](X1,X2) = 0
[SPLITD](X1,X2) = 2.X1 + 2.X2 + 1
[CASE1](X1,X2,X3) = X1 + X2 + 2.X3
[CASE2](X1,X2) = 2.X1 + X2
[UU1](X1,X2,X3) = 2.X1 + 2.X3 + 2
[UU2](X1,X2) = 1
[PRED](X) = 0
[CASE3](X1,X2) = 0
```

which is obtained by MU-TERM [17] proves DP-termination of  $U(\mathcal{R})$ .

Proofs of termination using the dependency pair approach are usually achieved by considering the *dependency graph*, or rather the *cycles* in the dependency graph.

**Theorem 2 (SCC termination [11])** *A TRS  $\mathcal{R}$  is terminating if and only if for all cycles  $\mathfrak{C}$  in the dependency graph there is an argument filtering  $\pi_{\mathfrak{C}}$  and a reduction pair  $(\succeq_{\mathfrak{C}}, \sqsupset_{\mathfrak{C}})$  such that  $\pi(\mathcal{R}) \subseteq \succeq_{\mathfrak{C}}$ ,  $\pi(\mathfrak{C}) \subseteq \succeq_{\mathfrak{C}} \cup \sqsupset_{\mathfrak{C}}$ , and  $\pi(\mathfrak{C}) \cap \sqsupset_{\mathfrak{C}} \neq \emptyset$ .*

In general, the dependency graph of a TRS is not computable and we need to use some approximation of it (e.g., the *estimated* dependency graph, see [1]). According to Theorem 2, the important point in proofs of SCC termination is the generation of appropriate argument filterings  $\pi_{\mathfrak{C}}$  and reduction pairs  $(\succeq_{\mathfrak{C}}, \sqsupset_{\mathfrak{C}})$  for each cycle  $\mathfrak{C}$  in the (estimated) dependency graph. In the following, we are interested in their generation by means of polynomial interpretations.

**Proposition 5** *Given a Core-SAFE program  $\mathcal{P}$ , there is a bijection between cycles in the dependency graph of the TRS  $\mathcal{R}_{\mathcal{P}} = U(\text{tr}P(\mathcal{P}))$  and recursive calls in  $\mathcal{P}$ .*

So, in our running example, the only existing cycle in the dependency graph contains the following dependency pairs:

```
SplitD(n,xs) -> Case1(n,n,xs)
Case1(s(x),n,xs) -> Case2(xs,n)
Case2(cons(x,xx),n) -> U1(n-1,x,xx)
U1(n',x,xx) -> SplitD(n',xx)
```

which corresponds to the internal recursive call of *splitD*. Here, we capitalize the first letter of a function name  $f$  to indicate its associated symbol  $f^{\sharp}$ . The problem now is that it is unclear how to give a suitable definition of *polynomial* associated to a given defined symbol  $f$ . In principle, a symbol  $f^{\sharp}$  can occur in *several* cycles  $\mathfrak{C}$ , thus leading to different polynomials  $\llbracket f^{\sharp} \rrbracket_{\mathfrak{C}}$ .

## 6 Case studies

We have applied Theorem 1 to the TRS's obtained by transforming the Core-Safe functions presented in Section 3—including function **length** with the obvious definition—and have obtained the polynomials shown in Figure 3.

The last one *insert* is the function inserting an element in a binary search tree. Its full-Safe definition is as follows:

Safe function	Polynomial inferred
<i>length</i> ( $x$ )	$2x + 1$
<i>splitD</i> ( $n, x$ )	$2n + 2x + 1$
<i>mergeD</i> ( $x, y$ )	$x + y + 1$
<i>msortD</i> ( $x$ )	No proof obtained
<i>insert</i> ( $x, t$ )	$2t + 1$

Figure 3: Polynomials obtained for several Core-Safe functions

```
data Tree a = Empty | Node (Tree a) a (Tree a)

insert x Empty = Node Empty x Empty
insert x (Node l y r)
  | x < y = Node (insert x l) y r
  | x == y = Node l y r
  | x > y = Node l y (insert x r)
```

and the CTRS resulting from the *trP* transformation is:

```
insert(x,t) -> case1(t,x)
case1(Empty,x) -> Node(10,x,r0)
  <= Empty -> 10,
  Empty -> r0
case1(Node(l,y,r),x) -> case2(c,l,y,r,x)
  <= lt(x,y) -> c
case2(False,l,y,r,x) -> case3(c',l,y,r,x)
  <= eq(x,y) -> c'
case2(True,l,y,r,x) -> Node(l',y,r)
  <= insert(x,l) -> l'
case3(False,l,y,r,x) -> case4(c'',l,y,r,x)
  <= gt(x,y) -> c''
case3(True,l,y,r,x) -> Node(l,y,r)
case4(False,l,y,r,x) -> error
case4(True,l,y,r,x) -> Node(l,y,r')
  <= insert(x,r) -> r'
```

From the above results, and interpreting the argument variables as characterizing the size of the corresponding data structures, we are glad to see that the bounds obtained are rather accurate. For instance, if an argument  $x$  is of type list and we interpret  $x$  as its length, the polynomial  $x + y + 1$  accurately bounds the number of recursive calls to *mergeD*( $x, y$ ). To see whether interpreting argument variables as sizes is correct or not we must pay attention to the interpretation given by Proposition 4 to data constructors. During the execution of a function  $f$ , the formal arguments of  $f$  will be replaced by actual ones and these consist just of ground terms formed by data constructors.

By knowing the polynomial interpretation obtained for these constructors, we can know the polynomial associated to the whole term representing the actual data structure passed to  $f$  as actual argument. In the examples above, we have obtained the following interpretation for the list data constructors:

$$\begin{aligned} \llbracket Nil \rrbracket &= 0 \\ \llbracket Cons(x, xs) \rrbracket &= k + xs \end{aligned}$$

being  $k = 1$  or  $k = 2$ . For binary trees, we have obtained:

$$\begin{aligned} \llbracket Empty \rrbracket &= 0 \\ \llbracket Node(l, x, r) \rrbracket &= 1 + l + r \end{aligned}$$

Then, the polynomial associated to a complete list will be related to its length and the one associated to a binary tree will coincide with its cardinality. This circumstance allows us to interpret argument variables as sizes (at least in the examples we have tried so far).

The polynomials obtained for *length* and *splitD* are less accurate, but at least they show an accurate linear dependency with their argument sizes. The bound for *insert* is also accurate as the binary tree needs not be balanced: in the worst case, the number of recursive calls grows linearly with the tree size.

We have not obtained a termination proof for *msortD*. We must be prepared for that due to the incompleteness of any termination proving algorithm. Apparently, the current TRS termination proving technology is not able to detect that the sizes of the lists passed as arguments to *msortD* in the two recursive calls are strictly smaller than the list of the external call. Manipulating the rules, it is possible to obtain a termination proof for  $U(trF(msortD))$  but it is not clear that this manipulation can always be obtained automatically.

## 7 Hierarchical composition of SAFE programs

When proving termination and complexity bounds of Safe programs, two strategies can be applied:

1. Either the whole program is transformed into a TRS, and then this is submitted

to a termination prover tool such as MUTERM.

2. Or else each function is separately analyzed for termination, assuming that the functions possibly called from the analyzed one in turn terminate.

Approach (1) is more realistic in the sense that the TRS exactly corresponds to the original Core-Safe program. In particular, constructor and function symbols are global to the whole program and the polynomials obtained for them, in case of success, guarantee that *every term* will be finitely rewritten. That is, every well-formed main expression using those symbols will terminate.

However, programs can be huge and the time needed by the termination tool will probably increase more than linearly with program size. So, it is worthwhile to investigate the modularity properties of the TRS obtained from the transformation of Safe programs. Intuitively, if we get a polynomial bounding the number of recursive calls of a particular function  $f$ , this is a property which depends on the *definition* of  $f$  (and, of the definition of all the functions used by  $f$ ) but not on its *use* in enclosing contexts. So, we expect that the polynomial of a function  $f$ , once obtained, will remain stable along the function definitions following that of  $f$  in the Safe text. In this case,  $f$ 's polynomial would not need to be inferred again when analyzing the functions that follows  $f$ .

Data constructors are a different matter as they miss a definition. Our experiments tell us that almost always they get a polynomial which clearly indicates the *size* of the data structure starting at this constructor, as it has been mentioned in Section 6. Then, we believe that it is desirable to force a fixed interpretation for the constructors, and this interpretation should convey the intuitive notion of size for the corresponding data structure.

In this vein, when inferring the polynomial for a particular function  $f$ , we could force the interpretation of the functions defined previously to  $f$ , to the polynomials obtained for them. In this way, the termination tool will have to infer only the polynomials for the new

defined symbols introduced by the transformation of  $f$ , and the amount of work would approximately be proportional to the complexity of  $f$ .

We should then prepare our termination tool for this mixed working mode, i.e. for receiving some polynomials as given and then inferring the rest.

## 8 Conclusions and Future Work

The preliminary experiments reported in this paper encourages us to continuing the exploration of the approach of using TRS termination tools to infer polynomial bounds on the number of recursive calls of real programs.

However much work remains to be done. In particular, we consider using DP-termination as a first approach to the problem due to the following reasons:

1. It forces *all* dependency pairs of the TRS to be strictly oriented while in fact only one strictly oriented pair per cycle is required for SCC-termination. The consequence is that termination proofs will fail more often.
2. Bounding the *total* number of recursive calls is sometimes a too rough bound on the length of recursive calls chains, due to the fact that a function may be multiple-recursive.

SCC-termination seems more promising in both respects but it requires a correct way of composing the polynomials obtained for the different cycles of the TRS. It is not clear that the least upper bound of all the polynomials gives always the correct bound. We conjecture that adding the polynomials obtained for a given symbol  $f^\#$  will always be a safe bound.

Another path to explore is to provide the termination proving tool with some help from the programmer in order to prove termination, and to infer correct polynomials, in cases such as the *msortD* function where the tool fails.

## References

- [1] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
- [2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, 1998.
- [3] G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Complexity classes and rewrite systems with polynomial interpretation. In G. Gottlob, E. Grandjean, and K. Seyr, editors, *12th International Workshop on Computer Science Logic, CSL '98*, volume 1584 of *Lecture Notes in Computer Science*, pages 372–384. Springer-Verlag, 1998.
- [4] G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11(1):33–53, 2001.
- [5] G. Bonfante, J.-Y. Marion, and J.Y. Moyén. Quasi-interpretations and Small Space Bounds. In J. Giesl, editor, *16th International Conference on Rewriting Techniques and Applications, RTA'05*, volume 3467 of *Lecture Notes in Computer Science*, pages 150–164. Springer-Verlag, 2005.
- [6] A. Cichon and P. Lescanne. Polynomial interpretations and the complexity of algorithms. In D. Kapur, editor, *11th International Conference on Automated Deduction, CADE'92*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 139–147. Springer-Verlag, 1992.
- [7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
- [8] E. Contejean, C. Marché, A.-P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):315–355, 2006.
- [9] N. Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.

- [10] F. Durán, S. Lucas, J. Meseguer, C. Marché, and X. Urbain. Proving operational termination of membership equational programs. *Higher-Order and Symbolic Computation*, page to appear, 2007.
- [11] J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34:21–58, 2002.
- [12] Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and René Thiemann. Automated termination analysis for haskell: From term rewriting to programming languages. In Frank Pfenning, editor, *RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 297–312. Springer, 2006.
- [13] D. Hofbauer. Termination Proofs by Context-Dependent Interpretations. In A. Middeldorp, editor, *12th International Conference on Rewriting Techniques and Applications, RTA'01*, volume 2051 of *Lecture Notes in Computer Science*, pages 108–121. Springer-Verlag, 2001.
- [14] D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In N. Dershowitz, editor, *3rd International Conference on Rewriting Techniques and Applications, RTA'89*, volume 355 of *Lecture Notes in Computer Science*, pages 167–177. Springer-Verlag, 1989.
- [15] D.S. Lankford. On proving term rewriting systems are noetherian. Technical report, Louisiana Technological University, 1979.
- [16] S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547–586, 2005.
- [17] Salvador Lucas. mu-term: A tool for proving termination of context-sensitive rewriting. In Vincent van Oostrom, editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2004.
- [18] M. Montenegro, R. Peña, and C. Segura. An inference algorithm for guaranteeing safe destruction. In *Proceedings of the 8th Symposium on Trends in Functional Programming, TFP'07, New York, April 2-4*, pages 1–16, Chapter XIV, 2007.
- [19] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.
- [20] R. Peña and C. Segura. A first-order functional language for reasoning about heap consumption. In *Proceedings of the 16th International Workshop on Implementation of Functional Languages, IFL'04*, pages 64–80, 2004.
- [21] R. Peña, C. Segura, and M. Montenegro. A sharing analysis for SAFE. In *Proceedings of the 7th Symposium on Trends in Functional Programming, TFP'06, Nottingham (UK), March 2006. Also in Selected papers of TFP'06, to be published by Intellect*, pages 205–221, 2006.