Rewriting Techniques for Analysing Termination and Complexity Bounds of *Safe* Programs*

Salvador Lucas

Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022
slucas@dsic.upv.es

Ricardo Peña

Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Prof. J. García Santesmases s/n, 28040
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 and stack 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 give 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 analysed by means of existing tools. Furthermore, we investigate how to use proofs of termination which combine the dependency pairs approach with polynomial interpretations to obtain suitable bounds to the length of chains of recursive calls in *Safe* programs.
Keywords: Termination, Term Rewriting Systems, Space complexity.

1 Introduction

Safe [21, 18] is a first-order eager functional language with facilities for programmer controlled destruction and copying of data structures, intended for compile time analysis of memory consumption. In *Safe*, the allocation and deallocation of compiler-defined memory regions for data structures are associated with function application. So, heap memory consumption depends both on the number of recursive calls and on the length of calls chains. In order to compute space bounds for the heap it is essential to compute bounds to these figures and, in turn, to previously ensure termination of such functions.

In this paper we investigate how to use rewriting techniques for proving termination of *Safe* programs and, at the same time, giving appropriate bounds to the number of recursive calls as a first step to compute space bounds. In particular, we introduce a transformation for proving termination of *Safe* programs by translating them into Term Rewriting Systems (TRS).

* Salvador Lucas was partially supported by the EU (FEDER) and the Spanish MEC grant TIN 2007-68093-C02-02. Ricardo Peña was partially supported by the Madrid Region Government under grant S-0505/TIC/0407 (PROMESAS).

Both termination and complexity bounds of programs have been investigated in the abstract framework of Term Rewriting Systems [3, 20]. A suitable way to prove termination of programs written in declarative programming languages like Haskell or Maude is translating them into (variants of) term rewriting systems and then using techniques and tools for proving termination of rewriting. See [9, 10] for recent proposals of concrete procedures and tools which apply to the aforementioned programming languages.

Polynomial interpretations have been extensively investigated as suitable tools to address different issues in term rewriting [3]. For instance, the limits of polynomial interpretations regarding their ability to prove termination of rewrite systems were first investigated in [12] by considering the *derivational 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 [11].

Complexity analysis of first order functional programs (or TRSs) has also been successfully addressed by using polynomial interpretations [4–6]. 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 [5] combine the use of *path orderings* [8] 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). Polynomials which are used in this setting are *weakly monotone*, i.e., if $x \geq y$ then $P(\ldots, x, \ldots) \geq P(\ldots, y, \ldots)$. 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 $P(\ldots, x, \ldots) > P(\ldots, y, \ldots)$). 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 [7, 17]. 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 $a_1 \; R \; a_2 \; R \; a_3 \cdots$. Throughout the paper, $\mathcal{X}$ denotes a countable set of variables and $\mathcal{F}$ denotes a signature, i.e., a set of function symbols $\{f, g, \ldots\}$, each having a fixed arity given by a mapping $ar : \mathcal{F} \to \mathbb{N}$. The set of terms built from $\mathcal{F}$ and $\mathcal{X}$ is $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Positions $p, q, \ldots$ are represented by chains of positive natural numbers used to address subterms of t. Positions are ordered by the standard prefix ordering $\leq$. The set of positions of a term t is

$\mathcal{P}os(t)$. The subterm at position p of t is denoted as $t|_p$ and $t[s]_p$ is the term t with the subterm at position p replaced by s. A *context* is a term $C[\,]$ with a 'hole' (formally, a fresh constant symbol). A rewrite rule is an ordered pair (l, r), written $l \to r$, 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 a TRS $\mathcal{R}$, a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ rewrites to s (at position $p \in \mathcal{P}os(t)$), written $t \to_{\mathcal{R}} s$, if there is a position $p \in \mathcal{P}os(t)$, a substitution σ, and a rule $l \to r$ in $\mathcal{R}$ such that $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. The term $t|_p$ is called a *redex* of t. A term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *innermost* rewrites to s, written $t \xrightarrow{i}_{\mathcal{R}} s$ if $t \to_{\mathcal{R}} s$ at position p and $t|_p$ contains no redex. A TRS $\mathcal{R}$ is (innermost) terminating if $\to_{\mathcal{R}}$ (resp. $\xrightarrow{i}_{\mathcal{R}}$) is terminating.

A *conditional, oriented* TRS (CTRS), has rules of the form $l \to r \Leftarrow C$, where $C = s_1 \to t_1, \ldots, s_k \to t_k$ is called an oriented condition. Given a CTRS $\mathcal{R}$, we let $\mathcal{R}_u$ be the set of rules $\mathcal{R}_u = \{l \to r \mid l \to r \Leftarrow C \in \mathcal{R}\}$. A CTRS which satisfies $Var(r) \subseteq Var(l) \cup Var(C)$ for every conditional rule is called a 3-CTRS. It is deterministic if the variables of the right-hand side t_i of every condition $s_i \to t_i$ of C are introduced before they are used in the left-hand side s_j of a subsequent condition $s_j \to t_j$. A deterministic 3-CTRS $\mathcal{R}$ is syntactically deterministic if, for every rule $l \to r \Leftarrow s_1 \to t_1, \ldots, s_k \to t_k$ in $\mathcal{R}$ every term t_i is a constructor term or a ground normal form with respect to $\mathcal{R}_u$.

3 The *Safe* language

Safe was introduced as a research platform to investigate analyses related to sharing of data structures and to memory consumption. Currently it is equipped with a type system guaranteeing that, in spite of the memory destruction facilities of the language, all well-typed programs will be free of dangling pointers at runtime. More information can be found at [21, 18] and [19].

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 differences are that *Safe* is eager and first-order. *Safe* admits two basic types (*booleans* and *integers*), algebraic datatypes (introduced by the usual **data** declarations), and the function definitions by means of conditional equations with the usual facilitites 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. Additionally, the programmer can specify a *destructive* pattern matching operation by using symbol ! after the pattern. The intended meaning is the destruction of the cell associated with the constructor symbol, thus allowing its reuse later. A *Safe* program consists of a sequence of (possibly recursive) function definitions together with a main expression.

The merge-sort program of Figure 1 uses a constant heap space to implement the sorting of the list. This is a consequence of the destructive constant-space versions *splitD* and *mergeD* of the funtions which respectively split a list into two pieces and merge two sorted lists. The types shown in the program are inferred by the compiler. A symbol ! in a type signature indicates that the corresponding

```
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
```

Fig. 1. Mergesort program in full-*Safe*, using constant heap space

data structure is destroyed by the function. A symbol ! in a righthand side variable expresses that a potentially condemned variable is reused. Variables ρ 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 has to be used for each construction. A function has one or more associated memory regions available for building constructions: a *working* region which can be addressed by using the reserved identifier *self* and a possibly empty collection of *output* regions which are passed as arguments. For this reason, the low-level syntax, called Core-*Safe* requires additional region arguments 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 region r. The compiler also *flattens* the expressions in such a way that applications of functions are made only to constants or to variables. 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 2. We use the notation $\overline{x_i}^n$ to abbreviate the sequence $x_1 \ldots x_n$.

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 variable pointing to a construction. We assume the existence of a heap and of a runtime environment, respectively mapping pointers to constructions and program variables to heap pointers. The complete operational semantics can be found in [21].

Function *splitD* defined in the *Safe* program of Figure 1 is translated into Core-*Safe* definition shown in Figure 3.

$$
\begin{array}{llll}
prog & \rightarrow & dec_1; \ldots; dec_n; e & \\
dec & \rightarrow & f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e & \{\text{recursive, polymorphic function}\} \\
e & \rightarrow & a & \{\text{atom: literal } c \text{ or variable } x\} \\
 & & \mid x@r & \{\text{copy}\} \\
 & & \mid x! & \{\text{reuse}\} \\
 & & \mid f\ \overline{a_i}^n\ @\ \overline{r_j}^l & \{\text{function application}\} \\
 & & \mid \textbf{let}\ x_1 = be\ \textbf{in}\ e & \{\text{non-recursive, monomorphic}\} \\
 & & \mid \textbf{case}\ x\ \textbf{of}\ \overline{alt_i}^n & \{\text{read-only case}\} \\
 & & \mid \textbf{case!}\ x\ \textbf{of}\ \overline{alt_i}^n & \{\text{destructive case}\} \\
alt & \rightarrow & C\ \overline{x_i}^n \rightarrow e & \\
be & \rightarrow & C\ \overline{a_i}^n\ @\ r & \{\text{constructor application}\} \\
 & & \mid e &
\end{array}
$$

Fig. 2. *Core-Safe* language definition

```
splitD n xs @ r1 r2 r3  = case n of
       0 -> let nil1 = []@r1 in let res1 = (nil1,xs!)@r3 in res1
       _ -> case! xs of
             []     -> let nil1 = []@r1 in let nil2 = []@r2 in
                          let res2 = (nil1,nil2)@r3 in res2
             : x xx -> let z = let n' = n-1 in splitD n' xx @ r1 r2 r3 in
                          let xs1 = case z of (ys1,ys2) -> ys1 in
                          let xs2 = case z of (zs1,zs2) -> zs2 in
                          let xs1' = (: x xs1)@r1 in
                          let res3 = (xs1', xs2)@r3 in res3
```

Fig. 3. Core-*Safe* version of *splitD*

4 Transformation from Core-*Safe* to CTRS

In this section we describe a transformation from Core-*Safe* programs to conditional term rewriting systems (CTRS). 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. In this way, variable, copy, and reuse expressions are collapsed into the variable expression. Also, the two variants of **case** are collapsed into one.

We assume that each **case** expression in a function definition has been labelled with a unique integer k. The transformation will be defined by using the following auxiliary functions:

1. *trP* takes a sequence of Core-*Safe* function definitions and returns a CTRS. Notice that the main expression is excluded.
2. *trF* takes a function definition and returns a set of conditional rewrite rules.
3. *trR* given an expression e (a binding expression be), the set V of its free variables, and a condition $C = s_1 \rightarrow t_1, \ldots, s_k \rightarrow t_k$ consisting of atomic (rewrite) conditions $s_i \rightarrow t_i$, returns the right-hand side of a rule together with its conditional part, and an additional, possibly empty, set of conditional rewrite rules. The condition C is treated as a list. If $C = [\,]$, then the generated right-hand side has no conditional part.
4. *trL* which, given an expression e and the set V of its free variables, yields the left part of a condition, and a sequence of atomic conditions to its left.

$$trP(\overline{def_i}^n) \stackrel{\text{def}}{=} \bigcup_{i=1}^n trF(def_i)$$
$$trF(f\ \overline{x_i}^n = e) \stackrel{\text{def}}{=} f(x_1,\ldots,x_n) \rightarrow trR(e, fv(e), [\,])$$
$$trR(c,V,C) \stackrel{\text{def}}{=} c \Leftarrow C$$
$$trR(x,V,C) \stackrel{\text{def}}{=} x \Leftarrow C$$
$$trR(C_r\ \overline{a_i}^n, V, C) \stackrel{\text{def}}{=} C_r(a_1,\ldots,a_n) \Leftarrow C$$
$$trR(f\ \overline{a_i}^n, V, C) \stackrel{\text{def}}{=} f(a_1,\ldots,a_n) \Leftarrow C$$
$$trR(k : \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^n, V, C) \stackrel{\text{def}}{=}$$
$$\{case_k(x, var(V)) \Leftarrow C\} \cup$$
$$\{case_k(C_i(x_{i1},\ldots,x_{in_i}), var(V)) \rightarrow trR(e_i, fv(e_i), [\,]) \mid i \in \{1..n\}\}$$
$$trR(\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, V, C) \stackrel{\text{def}}{=} trR(e_2, fv(e_2), C \mathbin{++} [, trL(e_1, fv(e_1)) \rightarrow x_1])$$
$$trL(e,V) \stackrel{\text{def}}{=} trR(e,V,[\,])\ \text{if}\ e \in \{c, x, C_r\ \overline{a_i}^n, f\ \overline{a_i}^n, \textbf{case}\}$$
$$trL(\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, V) \stackrel{\text{def}}{=} [trL(e_1, fv(e_1)) \rightarrow x_1,] \mathbin{++} trL(e_2, fv(e_2))$$

Fig. 4. Transformation from Core-*Safe* to CTRS

Let us assume that $var(V)$ assigns the variables in V to a given term t in a fixed ordering. The complete transformation is given in Figure 4. Our running example would be transformed into the following CTRS:

```
splitD(n,xs) -> case1(n,n,xs)
case1(0,n,xs) -> res1 <= Nil -> nil1, Tup(nil1,xs) -> res1
case1(S(x),n,xs) -> case2(xs,n)
case2(Nil,n) ->  res2 <= Nil -> nil1, Nil -> nil2, Tup(nil1,nil2) -> res2
case2(Cons(x,xx),n) -> res3 <= pred(n) -> n', splitD(n',xx) -> z,
                               case3(z) -> xs1, case4(z) -> xs2,
                               Cons(x,xs1) -> xs1', Tup(xs1',xs2) -> res3
case3(Tup(ys1,ys2)) -> ys1
case4(Tup(zs1,zs2)) -> zs2
```

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

Now we apply standard transformations from deterministic 3-CTRS to plain TRSs [20, Def.7.2.48]. If $\mathcal{R}$ is a 3-CTRS, let us call $U(\mathcal{R})$ to the resulting TRS. 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(pred(n),x,xx)
U1(n',x,xx) -> U2(splitD(n',xx),x)
U2(z,x) -> U3(case3(z),x,z)
U3(xs1,x,z) -> U4(case4(z),x,xs1)
U4(xs2,x,xs1) -> Tup(Cons(x,xs1),xs2)
case3(Tup(ys1,ys2)) -> ys1
case4(Tup(zs1,zs2)) -> zs2
```

In the following, let $\mathcal{R}_\mathcal{P}$ denote the system $U(trP(\mathcal{P}))$ resulting from applying the two aforementioned transformations to the Core-*Safe* program $\mathcal{P}$.

Proposition 2. *For every Core-Safe program $\mathcal{P}$, the TRS $\mathcal{R}_\mathcal{P}$ consists of nonoverlapping rules. Moreover, all the lefthand sides are of the form $f(p_1, \ldots, p_n)$ where the p_i are flat patterns.*

Proof. Straightforward by Proposition 1 and the U transformation. □

It is a standard result [20, Prop. 7.2.50] that the termination of $U(\mathcal{R})$ implies the termination of $\mathcal{R}$. Then, by Proposition 3, proving termination of $\mathcal{R}_\mathcal{P}$ implies the termination of the *Safe* program $\mathcal{P}$.

5 Termination of *Safe* programs

One of the main goals of this paper is providing suitable methods for proving termination of *Safe* programs. The following result shows that the transformation introduced in the previous section is appropriate for this goal: it preserves both termination and nontermination (i.e., characterizes termination) of *Safe* programs.

Proposition 3. *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 with e terminates in $\mathcal{R}$. Furthermore, in every term (except the last one, if it exists) of the reduction sequence of t_e there is only one innermost redex.*

It is well-known that the transformation U which has been used to obtain a TRS $U(\mathcal{R})$ from a deterministic 3-CTRS $\mathcal{R}$ is also nontermination preserving (see [20, Proposition 7.2.50]). Furthermore, for nonoverlapping, syntactically deterministic 3-CTRSs, termination of $\mathcal{R}$ and *innermost* termination of $U(\mathcal{R})$ are *equivalent* [20, Corollary 7.2.62]. According to Proposition 1, $trP(\mathcal{P})$ is a nonoverlapping, syntactically deterministic 3-CTRS for every Core-*Safe* program $\mathcal{P}$. Thus, by combining these facts, we can say the following.

Theorem 1. *A Core-Safe program $\mathcal{P}$, excluding its main expression, is terminating if and only if the TRS $U(trP(\mathcal{P}))$ is* innermost *terminating.*

Nowadays, several termination tools are able to prove (or disprove) innermost termination of rewriting automatically (e.g., AProVE, MU-TERM, TTT, etc.). Thanks to Theorem 1, they can be used now to prove termination (or nontermination!) of Core-*Safe* programs by using the transformation trP.

6 Dependency graph and recursive calls

Termination of (innermost) rewriting can be proved by using the dependency pairs approach [1]. Furthermore, our analysis of complexity bounds in Section

7 uses concepts coming from the dependency pairs approach. Thus, we briefly introduce and exemplify it in the following.

Given a TRS $\mathcal{R} = (\mathcal{C} \uplus \mathcal{D}, R)$ 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 $\mathcal{D} = \{root(l) \mid l \to r \in R\}$ and $\mathcal{C} = \mathcal{F} - \mathcal{D}$. The set $\mathsf{DP}(\mathcal{R})$ of *dependency pairs* for $\mathcal{R}$ is given as follows: if $f(t_1, \ldots, t_m) \to r \in R$ and $r = C[g(s_1, \ldots, s_n)]$ for some defined symbol $g \in \mathcal{D}$, and context $C[\cdot]$, and $s_1, \ldots, s_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, then $f^\sharp(t_1, \ldots, t_m) \to g^\sharp(s_1, \ldots, s_n) \in \mathsf{DP}(R)$, where $f^\sharp$ and $g^\sharp$ are new fresh symbols associated with f and g respectively.

Example 1. The dependency pairs which correspond to the TRS $\mathcal{R}_{SplitD}$ obtained at the end of Section 4 are the following (as usual, we capitalize –or duplicate– the first letter of a function name f to indicate its associated symbol $f^\sharp$.):

```
[1]  SPLITD(n,xs) -> CASE1(n,n,xs)
[2]  CASE1(S(x),n,xs) -> CASE2(xs,n)
[3]  CASE2(Cons(x,xx),n) -> UU1(pred(n),x,xx)
[4]  CASE2(Cons(x,xx),n) -> PRED(n)
[5]  UU1(n',x,xx) -> UU2(splitD(n',xx),x)
[6]  UU1(n',x,xx) -> SPLITD(n',xx)
[7]  UU2(z,x) -> UU3(case3(z),x,z)
[8]  UU2(z,x) -> CASE3(z)
[9]  UU3(xs1,x,z) -> UU4(case4(z),x,xs1)
[10] UU3(xs1,x,z) -> CASE4(z)
```

Termination of (innermost) rewriting is investigated by inspecting the *cycles* of the *dependency graph* $\mathsf{DG}(\mathcal{R})$ associated with the TRS $\mathcal{R}$. The nodes of the dependency graph are the dependency pairs $u \to v$ in $\mathsf{DP}(\mathcal{R})$; there is an arc from a node $u \to v$ to another node $u' \to v' \in \mathsf{DP}(\mathcal{R})$ if there are substitutions θ and θ' such that $\theta(v) \to^*_{\mathcal{R}} \theta'(u')$.

Remark 1. Proofs of termination of innermost rewriting using dependency pairs actually use an *innermost* dependency graph which is a subset of the standard one. In our context, though, both of them are identical due to the special shape of the rules in $\mathcal{R}_{\mathcal{P}}$. Thus, we will not further insist on that.

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]). Figure 5 shows the estimated dependency graph for $\mathcal{R}_{SplitD}$. Note that there is only *one* cycle: $\mathfrak{C} = \{1, 2, 3, 6\}$.

Due to the special structure of the rules in $\mathcal{R}_{\mathcal{P}}$ (see Proposition 2), it is clear that for every recursive call issued from $f(\delta_1, \ldots, \delta_n)$, where $\delta_1, \ldots, \delta_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ there is a *minimal* cycle in the dependency graph of $\mathcal{R}$ which contains a left-hand side $f(x_1, \ldots, x_n)$ thus closing a (minimal) cycle in the estimated dependency graph. Here, by a minimal cycle we mean a cycle which does not contain any proper subcycle.

Proposition 4. *Given a Core-Safe program $\mathcal{P}$, there is a bijection between* minimal *cycles in the dependency graph of the TRS $\mathcal{R}_{\mathcal{P}}$ and recursive calls in $\mathcal{P}$.*

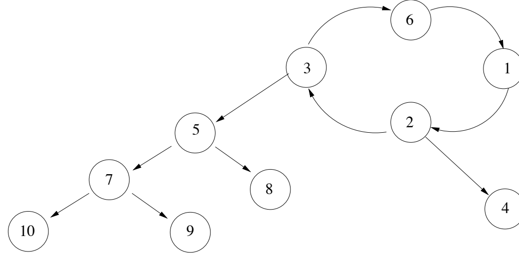

Fig. 5. Dependency graph for the transformed $\mathcal{R}_{SplitD}$

For instance, in our running example, the only existing cycle $\mathfrak{C}$ in the dependency graph contains the following dependency pairs:

```
[1]  SPLITD(n,xs) -> CASE1(n,n,xs)
[2]  CASE1(S(x),n,xs) -> CASE2(xs,n)
[3]  CASE2(Cons(x,xx),n) -> UU1(pred(n),x,xx)
[6]  UU1(n',x,xx) -> SPLITD(n',xx)
```

This cycle corresponds to the internal recursive call of *splitD*.

7 Explicit polynomial complexity bounds for *Safe*

The second main goal of this paper is developing methods for giving *explicit* complexity bounds to time/space consumption in *Safe* computations. Intuitively, a measure $[\![\ ]\!]$ aiming at associating a given complexity value to a particular function call $f(\delta_1, \ldots, \delta_k)$ for constructor terms $\delta_1, \ldots, \delta_k$ has to take into account the role of the arguments $\delta_1, \ldots, \delta_k$ in the computation of such value. Roughly speaking, we must associate a suitable k-ary *mapping* $[\![f]\!]$ to symbol f. In this paper we assume that $[\![f]\!]$ is a polynomial for all function symbols f.

In particular, $[\![f^\sharp]\!]$ is the polynomial interpreting the symbol $f^\sharp$ associated to the Core-*Safe* function symbol f. Moreover, assume that the dependency graph of $\mathcal{R}_{\mathcal{P}}$ contains only one cycle involving $f^\sharp$. Then, Theorem 2 below shows how (and when) the polynomial interpretation can be used to give explicit bounds to the number of calls to f in a given computation. First, we need to introduce some preliminary notions. Roughly speaking, the *usable rules* $\mathcal{U}(\mathcal{R}, \mathfrak{C})$ associated to a cycle $\mathfrak{C}$ in the dependency graph of $\mathcal{R}$ are obtained by first considering the rules $f(l_1, \ldots, l_n) \to r \in \mathcal{R}$ for all (unmarked, defined) symbols $f \in \mathcal{D}$ occurring in the right-hand sides v of the dependency pairs $u \to v \in \mathfrak{C}$ and then recursively adding the rules defining symbols in the right-hand sides of r [1, Definition 32]:

Definition 1 (Usable rules). *Let $\mathcal{R}$ be a TRS. For any symbol f let $Rules(\mathcal{R}, f)$ be the set of rules defining f and such that the left-hand side l has no redex as proper subterm. For any term t the set of* basic *usable rules $\mathcal{U}(\mathcal{R}, t)$ is as follows:*

$$\begin{aligned} \mathcal{U}(\mathcal{R}, x) &= \emptyset \\ \mathcal{U}(\mathcal{R}, f(t_1, \ldots, t_n)) &= Rules(\mathcal{R}, f) \cup \bigcup_{1 \le i \le ar(f)} \mathcal{U}(\mathcal{R}', t_i) \cup \bigcup_{l \to r \in Rules(\mathcal{R}, f)} \mathcal{U}(\mathcal{R}', r) \end{aligned}$$

where $\mathcal{R}' = \mathcal{R} - \mathit{Rules}(\mathcal{R}, f)$*. If* $\mathfrak{C} \subseteq \mathsf{DP}(\mathcal{R})$*, then* $\mathcal{U}(\mathcal{R}, \mathfrak{C}) = \bigcup_{l \to r \in \mathfrak{C}} \mathcal{U}(\mathcal{R}, r)$*.*

For instance, for cycle $\mathfrak{C}$ corresponding to our running example, the set of usable rules consists of a single rule: `pred(s(n)) -> n`. The following proposition shows why usable rules are interesting in our setting.

Proposition 5. *Let* $\mathcal{R}$ *be a TRS,* $t, s, u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$*, and* σ *be a substitution such that* $s = \sigma(t)$ *and* $\forall x \in \mathit{Var}(t)$*,* $\sigma(x)$ *is a normal form. Then,* $s \xrightarrow{i}^{*}_{\mathcal{R}} u$ *if and only if* $s \xrightarrow{i}^{*}_{\mathcal{U}(\mathcal{R},t)} u$*.*

The following theorem is the main result of this section. It shows that *explicit* polynomial bounds can be given to the number of function calls issued from a term $f(\delta_1, \ldots, \delta_n)$ where $\delta_1, \ldots, \delta_n$ are normal forms.

Theorem 2 (Explicit polynomial bounds). *Let* $\mathcal{R} = (\mathcal{F}, R)$ *be a TRS and* $f \in \mathcal{D}$ *be such that* $\mathsf{DG}(\mathcal{R})$ *contains only one cycle* $\mathfrak{C}$ *involving* $f^\sharp$*. Let* $[\![\ ]\!]$ *be a polynomial interpretation over the naturals satisfying that (1)* $[\![s]\!] \geq [\![t]\!]$ *for all* $s \to t \in \mathcal{U}(\mathcal{R}, \mathfrak{C}) \cup \mathfrak{C}$*; and (2)* $[\![u]\!] > [\![v]\!]$ *for at least one* $u \to v \in \mathfrak{C}$*. Let* $t = f(\delta_1, \ldots, \delta_n)$ *where* $\delta_1, \ldots, \delta_n$ *are normal forms.Then, the number* $N_f(t)$ *of calls to* f *during the innermost normalization of* t *is bounded by* $[\![f^\sharp(\delta_1, \ldots, \delta_n)]\!]$*:* $N_f(t) \leq [\![f^\sharp(\delta_1, \ldots, \delta_n)]\!] + 1$*.*

The polynomials which are necessary in Theorem 2 for obtaining polynomial bounds can be obtained in practice as part of the *innermost termination proof* for the TRS $\mathcal{R}$ using the dependency pairs approach which emphasizes the use of the dependency graph to obtain the proofs (DG-termination [1]). In our setting, we use the termination tool MU-TERM [16] to obtain the polynomial interpretations. For instance, consider the TRS $\mathcal{R}_{SplitD}$ obtained in Section 4 for our running example. The following polynomial intepretation:

```
[pred](X) = X            [case2](X1,X2) = 0       [case4](X) = 0
[S](X) = X               [Cons](X1,X2) = X2 + 1   [U5](X1,X2) = 0
[splitD](X1,X2) = 0      [U1](X1,X2,X3) = 0       [SPLITD](X1,X2) = X2 + 1
[case1](X1,X2,X3) = 0    [U2](X1,X2) = X1         [UU1](X1,X2,X3) = X3 + 1
[0] = 0                  [U3](X1,X2,X3) = 0       [CASE2](X1,X2) = X1
[Tup](X1,X2) = 0         [case3](X) = 0           [CASE1](X1,X2,X3) = X3 + 1
[Nil] = 0                [U4](X1,X2,X3) = 0
```

which is obtained by MU-TERM can be used in Theorem 2 to bound calls of the form $\mathtt{splitD}(\delta_1, \delta_2)$ for constructor terms δ_1, δ_2.

Remark 2. Although our results concern the use of polynomial interpretations over the naturals, they could be easily extended to a more general setting like the one described in [17] for polynomial interpretations over the *reals*.

The previous result can be further generalized to arbitrary terms as follows (the main difference with Theorem 2 is that *all* rules in the TRS $\mathcal{R}$ must be compatible with the ordering $\geq$ induced by the interpretation).

Corollary 1 (Explicit polynomial bounds II). *Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS and $f \in \mathcal{D}$ be such that $\mathsf{DG}(\mathcal{R})$ contains only one cycle $\mathfrak{C}$ involving $f^\sharp$. Let $[\![\ ]\!]$ be a polynomial interpretation over the naturals satisfying that (1) $[\![s]\!] \geq [\![t]\!]$ for all $s \to t \in \mathcal{R} \cup \mathfrak{C}$; and (2) $[\![u]\!] > [\![v]\!]$ for at least one $u \to v \in \mathfrak{C}$. Let $t = f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Then, the number $N_f(t)$ of* recursive *calls to f during the innermost normalization of t is bounded by $[\![f^\sharp(t_1, \ldots, t_n)]\!]$: $N_f(t) \leq [\![f^\sharp(t_1, \ldots, t_n)]\!] + 1$.*

7.1 Bounding the number of calls using the size of the arguments

Provided that the polynomial interpretation associated to the constructor symbols $c \in \mathcal{C}$ has the following shape:

$$[\![c]\!](x_1, \ldots, x_n) = c_1 x_1 + \cdots + c_n x_n + c_0$$

with $0 \leq c_i \leq 1$ for all i, $0 \leq i \leq n$, we obviously have that the size of a constructor term $\delta \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ is bounded from below by its interpretation, i.e., $|\delta| \geq [\![\delta]\!]$. Therefore, since $[\![f^\sharp]\!]$ has no negative coefficient and hence it is weakly monotone, we have that

$$1 + [\![f^\sharp]\!](|\delta_1|, \ldots, |\delta_n|) \geq 1 + [\![f^\sharp]\!]([\![\delta_1]\!], \ldots, [\![\delta_n]\!]) = 1 + [\![f^\sharp(\delta_1, \ldots, \delta_n)]\!] \geq N_f(t)$$

Furthermore, if constant constructor symbols c are interpreted by $[\![c]\!] = 1$; and for any other n-ary constructor symbol f (with $n > 0$) we have $c_i = 1$ for all i, $1 \leq i \leq n$ and $c_0 = 0$, then $|\delta| = [\![\delta]\!]$. Hence,

$$1 + [\![f^\sharp]\!](|\delta_1|, \ldots, |\delta_n|) = 1 + [\![f^\sharp(\delta_1, \ldots, \delta_n)]\!] \geq N_f(t)$$

i.e., we can think of the arguments $x_1, \ldots, x_n$ of the polynomial $[\![f^\sharp]\!](x_1, \ldots, x_n)$ as representing *sizes* of constructor terms and still giving appropriate bounds to the number of calls to f in any derivation from $f(\delta_1, \ldots, \delta_n)$ for terms δ_i of size x_i for each i, $1 \leq i \leq n$.

7.2 Space bounds and polynomial bounds

The relationship of the inferred polynomials with the space bounds we wish to infer is the following:

- Each function builds constructor cells in different heap regions which the compiler 'knows' because they are explicit in the text (they have been inferred in an early stage).
- The compiler infers an upper bound to the number of cells a single call to the function will build in each available region. This in general will be a polynomial because it may depend on calls to other functions. As these functions have already been inferred, the compiler knows the space costs charged by these functions to each region.
- Once we have the above, the function heap cost is obtained multiplying the (bound to the) number of recursive calls by the (bound to the) space cost of each call.

As the cell size is fixed for a given program, the compiler can compute an upper bound to the heap memory in terms of words or bytes. For stack consumption, the inference is even easier as the stack is not split into regions.

Safe **function**	**Polynomial inferred**	**Constructor interpretation**
$splitD(n, x)$	$x + 1$	$cons(y, ys) = ys + 1$
$mergeD(x, y)$	two cycles	
$length(x)$	x	$cons(y, ys) = ys + 1$
$append(x, y)$	x	$cons(y, ys) = ys + 1$
$insert(x, t)$	$t + 1$	$Node(t, x, t') = t + t' + 1$
$listInsert(x, y)$	y	$cons(y, ys) = ys + 1$
$insSortD(x)$	x	$cons(y, ys) = ys + 1$
$msortD(x)$	No proof obtained	
$mkTree(x)$	x	$cons(y, ys) = ys + 1$
$inorder(t)$	t	$Node(t, x, t') = t + t' + 1$

Fig. 6. Polynomials obtained for several Core-*Safe* functions

8 Case studies

We have applied the results in previous section to the TRS's obtained by transforming the Core-*Safe* functions presented in Section 3 and some other examples such as `length`, `append`, `insert`, `listInsert`, `insSort`, `mkTree`, `inorder`, which respectively gives the length of a list, appends two lists, inserts an element in a search tree, inserts an element in a sorted list, sorts a list by insertion, builds a search tree from a list, and does an inorder traversal of a tree, with the obvious definitions. We have obtained the polynomials shown in Figure 6.

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. According to Section 7.1, in order to see whether interpreting argument variables as sizes is correct or not we must pay attention to the interpretation given 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. By restricting the coefficients of these interpretations to 0 and 1 as explained in Section 7.1 we have obtained the interpretations shown in the table above. Then, the polynomial associated to a complete list is just its length and the one associated to a binary tree will coincides with its cardinality. This allows us to interpret argument variables as sizes.

The polynomial obtained for *length* is actually *exact*, the polynomial obtained for *splitD* is very accurate and shows the linear dependency with the size of the list: in the worst case, when the splitting position n exceeds the length of the list, *splitD* will be called as many times as the length of the list. 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 could not apply our results to *merge* because there are two minimal cycles which induce *different* polynomial interpretations for $merge^\sharp$: $[\![merge^\sharp]\!](x, y) = x$ and $[\![merge^\sharp]\!](x, y) = y + 1$, respectively. Since it is unclear (by now) how to combine different polynomial interpretations (in general), we cannot assert that, e.g., $[\![merge^\sharp]\!](x, y) = x + y + 1$.

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 that the list of the external call. Due to cases such as this, we plan to include in the source language the possibility of manually annotating the non-inferred functions with a polynomial.

9 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 it is submitted to a termination prover tool such as MU-TERM.
2. Or else, each function is separately analyzed for termination, assuming that the functions possibly called from the analized 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.

However, programs can be huge and the time needed by the termination tool will 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 global to the whole program. We believe that it would be desirable to *force* a fixed interpretation for them, automatically derived from the datatype definitions, conveying the intuitive notion of size for the corresponding data structure. Also, 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. So, the termination tool would infer only the polynomials for the new defined symbols.

Current TRS termination proving tools are not prepared for this mixed working mode in which some polynomials are forced and the rest are inferred. We are currently adapting our termination tool to this setting.

10 Related and Future Work

We have already cited in the introduction the works ([4–6]) aiming to classify TRS's in time and space complexity classes by using polynomial interpretations.

We make note that some results by these authors concern the computation of bounds for the size of the normal form term resulting from a rewriting sequence. In our context, it would be the size of the data structure returned by a function. This size is in principle not related to the heap space needed to compute the result, which is the topic of this paper. Closer to the research in this paper is the work about *derivation heights* by Hofbauer and others (see [12] and [11]). However, these works try to bound the *length* of rewriting sequences issued for terms in (polynomially) terminating TRSs. They pay no attention to the steps that correspond to particular symbols as done in this paper.

In the area of programming languages, there have been some attempts to infer complexity space bounds by using specialized type systems. The two following works compute linear space bounds of first order functional programs:

- Hughes and Pareto [14] incorporate in Embedded-ML the concept of region and their sized-types system is able to *type-check* heap and stack linear bounds from annotations given by the programmer.
- More recently, in a proof carrying code framework, Hofmann and Jost [13] have developed a type system to *infer* linear bounds on heap consumption. The underlying machinery is a Linear Programming system which solves the restrictions generated during type inference.

Related to the latter there has been the successful EU funded project *Mobile Resources Guarantees* [2] which, in addition to inferring space bounds, produces formal certificates of this property. These certificates can be verified by a proof-checker. A follow-on project is the Netherlands funded one AHA [22], which tries to extend the above results to space bounds beyond linear ones. Our approach seems promising with respect to these works in that any polynomial can be inferred by current termination proving tools.

The 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, bounding the number of recursive calls when several cycles are associated to the same symbol is an immediate goal. This corresponds either to multiple-recursive *Safe* functions or to single recursive ones with mutually exclusive calls. It is important to distinguish both cases when the goal is bounding the length of recursive calls chains.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
2. D. Aspinall, S. Gilmore, M. Hofmann, D. Sanella, and I. Stark. Mobile Resources Guarantees for Smart Devices. In *Proceedings of the Int. Workshop CASSIS'05*, pages 1–26. LNCS 3362, Springer, 2005.
3. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, Cambridge, 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. Moyen. Quasi-interpretations and Small Space Bounds. In J. Giesl, editor, *16th Int. Conf. on Rewriting Techniques and Applications, RTA'05*, volume 3467 of *LNCS*, 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. 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.
8. N. Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.
9. 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.
10. 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.
11. 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.
12. 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.
13. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL'03*, pages 185–197. ACM Press, 2003.
14. R. J. M. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space; Towards Embedded ML Programming. In *ICFP'99*, pages 70–81. ACM Press, 1999.
15. D.S. Lankford. On proving term rewriting systems are noetherian. Technical report, Louisiana Technological University, 1979.
16. S. 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.
17. S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547–586, 2005.
18. M. Montenegro, R. Peña, and C. Segura. A Type System for Safe Memory Management and its Proof of Correctness. In *PPDP'08*. ACM Press, 2008. To appear.
19. M. Montenegro, R. Peña, and C. Segura. An Inference Algorithm for Guaranteeing Safe Destruction. In *Accepted for presentation in LOPSTR'08, Valencia, Spain*, pages 1–15, July 2008.
20. E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.
21. R. Peña, C. Segura, and M. Montenegro. A Sharing Analysis for Safe. In *Proceedings of the Seventh Symposium on Trends in Functional Programming, TFP'06*, pages 205–221, 2006.
22. M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetsers. AHA: Amortized Space Usage Analysis. In *Proceedings of the 8th Symposium on Trends in Functional Programming, TFP'07, New York, April 2-4*, pages 1–16, Chapter XVI, 2007.