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 during the evaluation of an expression $f(e_1, \ldots, e_n)$ depends on the number of calls to f. Ensuring termination of *Safe* programs (or of particular function calls) is therefore essential to implement these analysis. Furthermore, being able to give bounds to the number of recursive calls required by a terminating initial call becomes essential in computing space bounds. In this paper, we first investigate how to analyze termination of *Safe* programs by using standard term rewriting techniques. First, we *transform* a *Safe* program into a term rewriting system. We provide a correct and *complete* transformation for that purpose, i.e., termination of the original *Safe* program $\mathcal{P}$ is completely *characterized* as (innermost) termination of the transformed TRS $\mathcal{R}_\mathcal{P}$. Then, termination can be automatically analysed by means of existing tools such as AProVE, MU-TERM, or TTT. We provide explicit bounds for the number of calls which are issued during the evaluation of an expression as above. We also investigate how to use proofs of termination which combine the dependency pairs approach with polynomial interpretations to obtain such numeric bounds.

Keywords: Termination, Term Rewriting Systems, Space complexity.

1 Introduction

Safe [23, 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* there is no garbage collector and the heap is split instead into disjoint *regions*. The allocation and deallocation of these compiler-defined regions is associated with function application. A function may charge space costs both to its own working region and to regions in scope passed as arguments. So, heap memory consumption depends critically on

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

the number of recursive calls deployed by a single external call to a function. In order to compute space bounds for the heap it is essential to compute a bound to this number, expressed as a function of the argument sizes.

In this paper we investigate how to prove termination of *Safe* programs and how to give appropriate bounds to the number of recursive calls as a first step to compute space bounds. Both termination and complexity bounds of programs have been investigated in the abstract framework of Term Rewriting Systems [3, 21]. 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 [10, 11] for recent proposals of concrete procedures and tools which apply to the aforementioned programming languages. Our first contribution is a transformation for proving termination of *Safe* programs $\mathcal{P}$ by translating them into Term Rewriting Systems (TRS).

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 were first investigated in [13] 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.

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* [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). In proofs of termination using the dependency pair approach [1], we can make a more flexible use of polynomials to 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. We investigate explicit bounds to the number of calls which are issued during the evaluation of an expression with respect to a *Safe* program $\mathcal{P}$. We show that they can be obtained by directly using the polynomial interpretations which are obtained in proofs of termination for the transformed TRS $\mathcal{R}_{\mathcal{P}}$ which combine the dependency pairs approach with polynomial interpretations to obtain such numeric bounds.

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 $\mathcal{T}(\mathcal{F}, \mathcal{X})$. 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 written $t \to_{\mathcal{R}} s$, if $t = C[\sigma(l)]$ and $s = C[\sigma(r)]$ for some context $C[\,]$, substitution σ, and rule $l \to r$ in $\mathcal{R}$. In the previous rewrite step, the term $\sigma(l)$ is called the *contracted 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 the redex contracted in the step $t \to s$ 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 [23, 18] and [19].

There are two versions of *Safe*: *full-Safe*, in which programmers write their programs, and *Core-Safe* (the internal version of *full-Safe*), in which program analyses are defined. Full-*Safe* syntax is close to Haskell's. *Safe* admits basic types, algebraic datatypes (introduced by the usual **data** declarations), and function definitions by means of conditional equations with the usual facilitites for pattern matching, **let** and **case** expressions, and **where** clauses. Additionally, the programmer can specify a *destructive* pattern matching operation by using the symbol ! after the pattern. The intended meaning is the destruction of the cell associated with the constructor, thus allowing its reuse. A *Safe* program consists of a sequence of (possibly recursive) function definitions together with a main expression. For the moment, mutual recursion is not allowed.

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 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'.

```
splitD :: ∀aρ1ρ2ρ3.Int → [a]!@ρ1 → ρ1 → ρ2 → ρ3 → ([a]@ρ2, [a]@ρ1)@ρ3
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

```
prog → dec1; ...; decn; e
dec  →   f x̄ᵢⁿ @ r̄ⱼˡ = e   {recursive, polymorphic function}
e    →   a                 {atom: literal c or variable x}
       | x@r               {copy}
       | x!                {reuse}
       | f āᵢⁿ @ r̄ⱼˡ        {function application}
       | let x1 = be in e  {non-recursive, monomorphic}
       | case x of ālt_iⁿ  {read-only case}
       | case! x of ālt_iⁿ {destructive case}
alt  → C x̄ᵢⁿ → e
be   →  C āᵢⁿ @ r          {constructor application}
       | e
```

Fig. 2. *Core-Safe* language definition

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 may build constructions in several regions: a *working* region, addressed by using the reserved identifier *self*, and a possibly empty collection of *output* regions which are passed as arguments. For this reason, the Core-*Safe* syntax requires additional region arguments both in 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.

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

```
splitD n xs @ r1 r2 r3  = case n of
       0 -> let nil1 = []@r2 in let res1 = (nil1,xs!)@r3 in res1
       _ -> case! xs of
             []     -> let nil1 = []@r2 in let nil2 = []@r1 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)@r2 in
                       let res3 = (xs1', xs2)@r3 in res3
```

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

of a runtime environment, respectively mapping pointers to constructions and program variables to heap pointers. The complete operational semantics can be found in [23].

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

4 Transformation from Core-*Safe* to CTRS

In this section we describe a transformation from Core-*Safe* programs to conditional term rewriting systems (CTRS). We can even simplify the Core-*Safe* syntax, because destructive patterns and regions are not relevant for termination purposes. In this way, variable, copy, and reuse expressions are collapsed into the variable expression, and 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 $C = s_1 \to t_1, \ldots, s_k \to t_k$ consisting of atomic (rewrite) conditions $s_i \to t_i$, 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.

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,
```

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

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

```
                                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 [21, 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 non-overlapping rules. Moreover,*

1. *All the left-hand sides are of the form $f(p_1,\ldots,p_n)$ where (1) all p_i are variables if f is a function symbol defined in $\mathcal{P}$ or (2) they are flat patterns otherwise.*
2. *The right-hand sides are of the form $g(e_1,\ldots,e_m)$ with g being a function symbol, all e_i, $1 < i \leq m$ are variables and e_1 is either a variable, a flat pattern, or a term $f(a_1,\ldots,a_n)$ with f being a function symbol and the a_j variables or basic constants.*

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

5 Termination of *Safe* programs

The following result shows that the transformation introduced in the previous section 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 [21, Proposition 7.2.50]). Furthermore, for nonoverlapping, syntactically deterministic 3-CTRSs, termination of $\mathcal{R}$ and *innermost* termination of $U(\mathcal{R})$ are *equivalent* [21, 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 8 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 $f^\sharp$.):

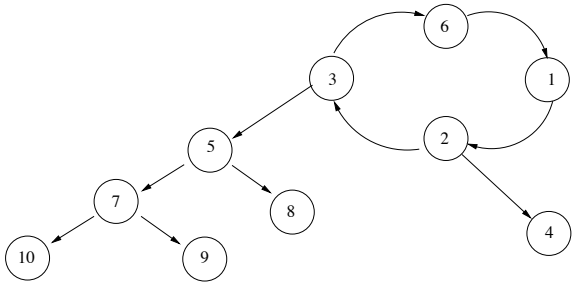

Fig. 5. Dependency graph for the transformed $\mathcal{R}_{SplitD}$

```
[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[1] 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 $\theta(v) \to^*_{\mathcal{R}} \theta'(u')$. 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: $\mathfrak{C} = \{1, 2, 3, 6\}$.

6.1 Dependency graph for $\mathcal{R}_{\mathcal{P}}$

Due to the special structure of the rules in $\mathcal{R}_{\mathcal{P}}$ (see Proposition 2), it is clear that, given a defined symbol f in the original SAFE program $\mathcal{P}$:

1. There is at most one rule $u \to v \in \mathsf{DP}(\mathcal{R}_{\mathcal{P}})$ such that $root(u) = f^\sharp$. Furthermore, $u = f^\sharp(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are variables.
2. If $u \to v \in \mathsf{DP}(\mathcal{R}_{\mathcal{P}})$ and v contains an occurrence of f or $f^\sharp$, then either $v = f^\sharp(v_1, \ldots, v_n)$ or $v = g^\sharp(f(v_{11}, \ldots, v_{1n}), v_2, \ldots, v_m)$ for some defined symbol g.

Thus, 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

[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 $\mathcal{R}_{\mathcal{P}}$.

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 $\mathcal{P}$.*

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*.

In the following, given a defined symbol f from the original SAFE program $\mathcal{P}$, we let $\mathsf{DG}(\mathcal{R}_\mathcal{P}, f^\sharp)$ be the subgraph of $\mathsf{DG}(\mathcal{R}_\mathcal{P})$ which contains all (and only) the minimal cycles in $\mathsf{DG}(\mathcal{R}_\mathcal{P})$ starting from $f^\sharp(x_1, \ldots, x_n) \to v$.

Definition 1 (Nested cycles). *Given a defined symbol f from a SAFE program $\mathcal{P}$, we say that a minimal cycle $\mathfrak{C}$ in $\mathsf{DG}(\mathcal{R}_\mathcal{P}, f^\sharp)$ contains* nested calls to other cycles *if there is $u \to v \in \mathfrak{C}$ such that f occurs in v (hence* below *the root of v). The* nesting degree $\mathsf{nd}(\mathfrak{C}, f^\sharp)$ *of $\mathfrak{C}$ in $\mathsf{DG}(\mathcal{R}_\mathcal{P}, f^\sharp)$ is the number of such occurrences in $\mathfrak{C}$. The* nesting degree $\mathsf{ND}(\mathcal{R}_\mathcal{P}, f^\sharp)$ *of $\mathsf{DG}(\mathcal{R}_\mathcal{P}, f^\sharp)$ is the maximum of $\mathsf{nd}(\mathfrak{C}, f^\sharp)$ for the minimal cycles $\mathfrak{C}$ in $\mathsf{DG}(\mathcal{R}_\mathcal{P}, f^\sharp)$.*

For instance, the nesting degree of the cycle $\mathfrak{C}$ in our running example is 0. Consider now the following SAFE program *FibN* encoding Fibonacci's function using Peano's natural numbers:

```
add Zero y = y
add (Suc x) y = Suc (add x y)

fibN Zero          = Suc (Zero)
fibN (Suc (Zero))  = Suc (Zero)
fibN (Suc (Suc x)) = add (fibN (Suc x)) (fibN x)
```

and the transformed TRS $\mathcal{R}_{FibN}$

```
add(x1,x2) -> case1(x2,x1)
case1(Suc(x3),x1) -> flat1(add(x1,x3))
flat1(x4) -> Suc(x4)
case1(Zero,x1) -> x1
fibN(x1) -> case2(x1)
case2(Suc(x2)) -> case3(x2)
case2(Zero) -> Suc(Zero)
case3(Suc(x3)) -> flat3(fibN(Suc(x3)),x3)
flat3(x4,x3) -> flat2(fibN(x3),x4)
flat2(x5,x4) -> add(x4,x5)
case3(Zero) -> Suc(Zero)
```

Note that $\mathsf{DG}(\mathcal{R}_{FibN}, fibN)$ consists of two minimal cycles. Cycle $\mathfrak{C}_1$ has the following dependency pairs:

```
FIBN(x1) -> CASE2(x1)          CASE3(Suc(x3)) -> FLAT3(fibN(Suc(x3)),x3)
CASE2(Suc(x2)) -> CASE3(x2)    FLAT3(x4,x3) -> FIBN(x3)
```

Its nesting degree is 1: $\mathsf{nd}(\mathfrak{C}_1, FIBN) = 1$. The second minimal cycle $\mathfrak{C}_2$ consists of the following dependency pairs:

```
FIBN(x1) -> CASE2(x1)                  CASE3(Suc(x3)) ->FIBN(Suc(x3))
CASE2(Suc(x2)) -> CASE3(x2)
```

Its nesting degree is 0: $\mathsf{nd}(\mathfrak{C}_2, FIBN) = 0$.

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 with non-negative integer coefficients 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. Then, Theorems 2 and 3 below show 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 2 (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:*

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

where $\mathcal{R}' = \mathcal{R} - Rules(\mathcal{R}, f)$. 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 *SplitD*, 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, $s, t, u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, and σ be a substitution such that $s = \sigma(t)$ and $\forall x \in Var(t)$, $\sigma(x)$ is a normal form. Then, $s \xrightarrow{i}{}^*_{\mathcal{R}} u$ if and only if $s \xrightarrow{i}{}^*_{\mathcal{U}(\mathcal{R}, t)} u$.*

More refined notions of usable rules for innermost rewriting have been recently introduced in [12, Definition 15]. Indeed, they could be used instead of the ones in Definition 2 because Proposition 5 also holds for them.

7.1 No nested cycles in the graph

In the following, the number of calls to f during the innermost normalization of a term t is denoted by $N_f(t)$.

Theorem 2 (Explicit polynomial bounds I). *Let $\mathcal{P}$ be a SAFE program, $\mathcal{R} = (\mathcal{F}, R)$ be the transformed TRS (i.e., $\mathcal{R} = \mathcal{R}_\mathcal{P}$), and $f \in \mathcal{D}$ be defined*

in $\mathcal{P}$ and such that $\mathsf{ND}(\mathcal{R}_\mathcal{P}, f^\sharp) = 0$. Let $[\![\ ]\!]$ be a polynomial interpretation over the naturals satisfying that, for all minimal cycles $\mathfrak{C}$ in $\mathsf{DG}(\mathcal{R}_\mathcal{P}, f^\sharp)$: (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, $N_f(t) \leq [\![f^\sharp(\delta_1, \ldots, \delta_n)]\!] + 1$.

Remark 1. Note that Theorem 2 provides a *concrete, numeric* bound to $N_f(t)$. This is in contrast to the related works we are aware of, which rather provide a bound in $O(P)$ for some polynomial P (in the *size* of the terms), i.e., the constant C which multiplies P is usually not provided.

Many times, 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]). 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 [17] 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.

7.2 Nested cycles in the graph

Theorem 2 can be used to give explicit bounds to the number of calls to function symbols f whose 'local' dependency graph $\mathsf{DG}(\mathcal{R}_\mathcal{P}, f^\sharp)$ has no nested cycles. When nested cycles are present, we have to pay attention not only to the interpretations but also to the combinatorial structure of the cycles.

Theorem 3 (Explicit polynomial bounds II). *Let $\mathcal{P}$ be a SAFE program, $\mathcal{R} = (\mathcal{F}, R)$ be the transformed TRS (i.e., $\mathcal{R} = \mathcal{R}_\mathcal{P}$), and $f \in \mathcal{D}$ be defined in $\mathcal{P}$ and such that $d = \mathsf{ND}(\mathcal{R}, f^\sharp) > 0$. Let $[\![\ ]\!]$ be a polynomial interpretation over the naturals satisfying that, for all minimal cycles $\mathfrak{C}$ in $\mathsf{DG}(\mathcal{R}, f^\sharp)$, (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, $N_f(t) < (d+1)^{[\![f^\sharp(\delta_1, \ldots, \delta_n)]\!]+1}$.*

The SAFE program *FibN* above shows that Theorem 3 is really necessary because Theorem 2 does not properly bound the number of calls to symbols f with nested cycles: it is clear that a call $FibN(n)$ for a given number n (in Peano's notation) leads to an exponential number of calls to f which would not be captured by using Theorem 2.

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

The size $|t|$ of a term t is the number symbols occurring in t: $|t| = 1$ if t is a variable or a constant, and $|f(t_1, \ldots, t_n)| = 1 + \sum_{i=1}^{n} |t_i|$. If the polynomials 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, $0 \leq i \leq n$, then the size $|\delta|$ of a constructor term $\delta \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ is bounded from below by its interpretation: $|\delta| \geq [\![\delta]\!]$. Therefore, since $[\![f^\sharp]\!]$ has no negative coefficient, we have, for $t = f(\delta_1, \ldots, \delta_n)$:

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

Thus, the arguments $x_1, \ldots, x_n$ of the polynomial $[\![f^\sharp]\!](x_1, \ldots, x_n)$ can be thought of as *sizes* of constructor terms passed as arguments to f.

7.4 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 Core-*Safe* text.
- The compiler infers an upper bound to the number of cells a single call to the function will build in each region in scope. 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. For instance, in the *splitD* example of Sec. 3.1, the space inferred for a single recursive call is $\{\rho_1 \mapsto -1, \rho_2 \mapsto 1, \rho_3 \mapsto 0\}$, while an upper bound for the non-recursive cases is $\{\rho_1 \mapsto 1, \rho_2 \mapsto 1, \rho_3 \mapsto 1\}$.
- Once we have the above, the function heap cost is obtained by multiplying the (bound to the) number of recursive calls by the (bound to the) space cost of each call, and then adding the space cost of the non-recursive cases. In a call $splitD(n, x)$, this gives $\{\rho_1 \mapsto -x, \rho_2 \mapsto x + 2, \rho_3 \mapsto 1\}$

In total, a positive balance of 3 cells is obtained, which confirms that *splitD* runs in constant heap space.

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 as a function on the input sizes. For stack consumption, the inference is even easier as the stack is not split into regions.

8 Implementation of the analysis

As remarked above, the polynomials which are necessary in Theorems 2 and 3 above can be obtained in practice by using standard tools for proving termination of innermost rewriting like AProVE[2], MU-TERM[3], or TTT[4]. Actually, all these

[2] `http://aprove.informatik.rwth-aachen.de`
[3] `http://zenon.dsic.upv.es/muterm`
[4] `http://colo6-c703.uibk.ac.at/ttt`

tools are able to compute the dependency graph of a TRS and can also obtain polynomial interpretations which are compatible with the (usable) rules and dependency pairs of the cycles in the graph.

Nevertheless, it is worth noticing that, in many cases, the polynomials obtained by such tools do *not* fit the requirements in Theorems 2 or 3. In particular, since the main goal of these tools is proving termination, the polynomial interpretations which are obtained can be *different* for *different* cycles. Also, most of them use *argument filterings* [20] which often remove parts of the rules and dependency pairs. This makes the obtention of a termination proof easier. However, some calls to the considered function f can be lost due to the removal of parts of the rules. Clearly, this would lead to wrong conclusions[5].

Another possibility is the direct implementation of Theorems 2 or 3. Borrowing the well-known procedures for solving inequalities like $s \geq t$ or $u > v$, for terms s, t, u, v, by using (parametric) polynomial interpretations over the naturals (see [7] for a recent account), we could obtain the required interpretations which could then be safely used in the corresponding theorems. By lack of space, we cannot develop this further.

9 Case studies

We have applied our results 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. By using different termination tools, we have obtained the polynomials shown in Figure 6. Complete details about our experiments and about the *Safe*compiler can be found at:

`http://www.dsic.upv.es/~slucas/papers/lopstr08/experiments`

`http://dalila.sip.ucm.es/~ricardo`

From the above results, we are glad to see that the bounds obtained are rather accurate: the polynomial obtained for *length* is actually *exact*, and the one obtained for *splitD* is very accurate. The bound for *insert* is also accurate as the binary tree needs not be balanced. In the case of *merge* there are two minimal cycles with nesting degree zero. The tool infers $[\![merge^\sharp]\!](x, y) = x + y + 1$ which makes both cycles to decrease, as required by Theorem 2. Finally, the following *Safe* program has a cuadratic number of calls:

```
inter xs ys = inter2 xs ys ys
inter2 [] ys1 ys2   = []
inter2 (x:xs) [] ys = inter2 xs ys ys

inter2 (x:xs) (y:ys) ys2
   | x == y    = x : inter2 xs ys ys2
   | x /= y    = inter2 (x:xs) ys ys2
```

[5] The 2006 standalone version of the tool AProVE (AProVE 1.2), though, offers the possibility of *disabling* many of these features.

Safe **function**	**Polynomial inferred**	**Constructor interpretation**
$splitD(n, x)$	$x + 1$	$cons(y, ys) = ys + 1$
$mergeD(x, y)$	$x + y + 1$	$cons(n, y) = y + 1$
$length(x)$	x	$cons(y, ys) = ys + 1$
$append(x, y)$	x	$cons(y, ys) = ys + 1$
$insert(x, t)$	$t + 1$	$Node(t, x, t') = t + t' + 1$
$inter2(x, y, z)$	$xz + 2y + 2$	$cons(y, ys) = 2y + ys + 2$
$listInsert(x, y)$	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

which are captured by the polynomial interpretation computed by the tool: $inter2(x, y, z) = xz + 2y + 2$.

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.

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, this would correspond to 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 [13]) 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 functional 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 [15] incorporated 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.
- Hofmann and Jost [14] developed a type system *inferring* linear bounds on heap consumption, being the underlying machinery a Linear Programming system solving 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 which can be verified by a proof-checker. A follow-on project is the Netherlands funded one AHA [24], 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.

In the logic programming field, there have been also several approaches for inferring complexity costs (mainly time costs). We mention [16] and [8], both based on abstract interpretation. The first one generates a set of linear constraints, so it cannot infer higher-degree polynomials.

Our results giving explicit bounds on the number of recursive calls are completely general and could be directly applied to any other eager first-order functional language. 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 calls of real programs.

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. *Jour. Funct. 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 *LNAI*, 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. S. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *PLDI'90, White Plains, New York.* ACM, 1990.
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, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In F. Pfenning, editor, *RTA*, volume 4098 of *LNCS*, pages 297–312. Springer, 2006.
12. J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and Disproving Termination of Higher-Order Functions. In B. Gramlich, editor, *Int. Work. on Frontiers of Combining Systems, FroCoS'05*, LNAI 3717, pages 216–231. Springer, 2005.

13. D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In N. Dershowitz, editor, *Int. Conf. on Rewriting Techniques and Applications, RTA'89*, volume 355 of *LNCS*, pages 167–177. Springer, 1989.
14. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. 30th ACM Symp. on Principles of Programming Languages, POPL'03*, pages 185–197. ACM Press, 2003.
15. R. J. M. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space; Towards Embedded ML Programming. In *Proc. 4th ACM Int. Conf. on Funct. Programming, ICFP'99*, pages 70–81, Paris, France, Sept. 1999.
16. A. King, K. Shen, and F. Benoy. Lower-bound time-complexity analysis of logic programs. In *ILPS*, pages 261–275, 1997.
17. S. Lucas. MU-TERM: A Tool for Proving Termination of Context-Sensitive Rewriting. In *RTA'04, V. van Oostrom (ed.), LNCS 3091, Springer*, pages 200–209, 2004.
18. M. Montenegro, R. Peña, and C. Segura. A Type System for Safe Memory Management and its Proof of Correctness. In *ACM Principles and Practice of Declarative Programming, PPDP'08, Valencia, Spain, July. 2008*, pages 152–162, 2008.
19. M. Montenegro, R. Peña, and C. Segura. An Inference Algorithm for Guaranteeing Safe Destruction, 2008. *Presented in LOPSTR'08, Valencia, Spain.*
20. G. Nadathur, editor. *Princ. and Pract. of Declarative Programming, Int. Conf. PPDP'99, Paris, France, Sep. 29 - Oct. 1*, LNCS 1702. Springer, 1999.
21. E. Ohlebusch. *Advanced Topics in Term Rewriting.* Springer, 2002.
22. R. Peña and C. Segura. Formally Deriving a Compiler for SAFE. In *Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages, IFL'06, Budapest, Hungary, Sept. 2006*, pages 429–446, 2006.
23. R. Peña, C. Segura, and M. Montenegro. A Sharing Analysis for SAFE. In *Selected Papers of the 7th Symp. on Trends in Functional Programming, TFP'06.*, pages 109–128. Intellect, 2007.
24. M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetsers. AHA: Amortized Space Usage Analysis. In *Proc. Symp. on Trends in Functional Programming, TFP'07, New York, April 2-4*, pages XVI, 1–16, 2007.

Appendix: Proof of Theorems

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.*

Proof. The resulting system is an oriented CTRS just by inspection of the generated rules.

For every defined symbol f, a single rule is generated with all argument variables $\overline{x_i}^n$ distinct. For every **case** expression labelled k, a non-overlapping set of rules $case_K$, one for every data constructor C_i, is generated. Each rule introduces distinct pattern variables $\overline{x_{ij}}^{n_i}$. So, the CTRS is non-overlapping and left-linear.

By induction on the calls to $trR(e, V, C)$, it is easy to show that $fv(e) \subseteq V \cup var(C)$. From here, and by inspection of the rules $l \to r \Leftarrow C$ generated, we conclude that the system is 3-CTRS. Finally, the last rule of trL satisfies $fv(e_2) \subseteq V \cup \{x_1\}$. By induction on the number of simple conditions included in C we can prove that every condition C is syntactically deterministic. □

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 to 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.*

Proof. It simultaneously uses the small-step semantics of SAFE (not shown in this paper, see [22]) and the CTRS rules. It proceeds by induction on the depth k of the definition of the function symbols in the initial term. Then, by cases on the expressions e in function's bodies and, when the expression e is a **let** or a **case**, by induction on the number of operational semantics steps reducing e to normal form. □

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 $\mathcal{P}$.*

Proof. Straightforward by inspection of the transformation and by knowing that the arcs in the estimated dependency graph require unification between the right part of a dependency pair and the left part of another pair. In our transformed system, a cycle is closed when the internal call $f^\sharp(t_1, \ldots, t_n)$ to a recursive Core-SAFE function f unifies with the dependency pair $f^\sharp(x_1, \ldots, x_n) \to r$ coming from the initial (and only) rule defining function f. □

In the following proof, we use some additional notation. Positions $p, q, \ldots$ are represented by chains of positive natural numbers used to address subterms of

t. The set of positions of a term t is $\mathcal{P}os(t)$. Positions of non-variable symbols in t are denoted as $\mathcal{P}os_{\mathcal{F}}(t)$, and $\mathcal{P}os_{\mathcal{X}}(t)$ are the positions of variables.

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

Proof. The *if* part is obvious. For the *only if* part, we proceed by induction on the length of the sequence $s \xrightarrow{i}{}^*_{\mathcal{R}} u$. If $s = \sigma(t) = u$, it is trivial. Otherwise, if $s \xrightarrow{i}_{\mathcal{R}} s' \xrightarrow{i}{}^*_{\mathcal{R}} u$, then there is a substitution θ, a position $p \in \mathcal{P}os(s)$ and a rule $l \to r \in \mathcal{R}$ such that $s|_p = \theta(l)$ and $s'|_p = \theta(r)$. Furthermore, since $\sigma(x)$ is a normal form for all $x \in Var(t)$, p is a nonvariable position of t: $p \in \mathcal{P}os_{\mathcal{F}}(t)$. Hence, according to Definition 2, $l \to r \in \mathcal{U}(\mathcal{R}, t)$ and $s \xrightarrow{i}_{\mathcal{U}(\mathcal{R},t)} s'$. Furthermore, we can write $s' = \sigma'(t')$ for $t' = t[r]_p$, where $\sigma'(x) = \sigma(x)$ for all $x \in Var(t)$ and $\sigma'(x) = \theta(x)$ for all $x \in Var(r)$ (as usual, we assume that $Var(t) \cap Var(l \to r) = \emptyset$). Furthermore, since $s|_p$ is an *innermost* redex, we can assume that $\theta(x)$ is a normal form for all $x \in Var(r)$. Hence, $\sigma'(x)$ is a normal form for all $x \in Var(t')$. By the induction hypothesis we know that $s' \xrightarrow{i}{}^*_{\mathcal{U}(\mathcal{R},t')} u$. Since $\mathcal{U}(\mathcal{R}, t') \subseteq \mathcal{U}(\mathcal{R}, t)$, we have $s \xrightarrow{i}{}^*_{\mathcal{U}(\mathcal{R},t)} s' \xrightarrow{i}{}^*_{\mathcal{U}(\mathcal{R},t)} u$ as desired. □

Theorem 2 *Let $\mathcal{P}$ be a SAFE program, $\mathcal{R} = (\mathcal{F}, R)$ be the transformed TRS (i.e., $\mathcal{R} = \mathcal{R}_{\mathcal{P}}$), and $f \in \mathcal{D}$ be defined in $\mathcal{P}$ and such that $\mathsf{ND}(\mathcal{R}_{\mathcal{P}}, f^\sharp) = 0$. Let $[\![\ ]\!]$ be a polynomial interpretation over the naturals satisfying that, for all minimal cycles $\mathfrak{C}$ in $\mathsf{DG}(\mathcal{R}_{\mathcal{P}}, f^\sharp)$: (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, $N_f(t) \leq [\![f^\sharp(\delta_1, \ldots, \delta_n)]\!] + 1$.*

Proof. By induction of the number $N_f(t)$ of calls to f in the sequence. If $N_f(t) = 1$, then since all polynomials contain non-negative coefficients only, we have that $[\![f^\sharp(\delta_1, \ldots, \delta_n)]\!] \geq 0$; hence $[\![f^\sharp(\delta_1, \ldots, \delta_n)]\!] + 1 \geq N_f(t)$. If $N_f(t) > 1$, then, without loosing generality, considering that the evaluation of terms proceeds according to an innermost strategy, we can write the sequence as follows:

$$t = \underline{f(\delta_1, \ldots, \delta_n)} \xrightarrow{i}{}^+_{\mathcal{R}} C[f(w'_1, \ldots, w'_n)] \xrightarrow{i}{}^*_{\mathcal{R}} C[f(\delta'_1, \ldots, \delta'_n)] \xrightarrow{i}{}^+_{\mathcal{R}} \delta$$

for some context $C[\,]$, terms $w'_1, \ldots, w'_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, and normal forms $\delta'_1, \ldots, \delta'_n$. Let us further develop the sequence from t to $C[f(w'_1, \ldots, w'_n)]$ as follows

$$t = f(\delta_1, \ldots, \delta_n) = t_1 \xrightarrow{i}_{\mathcal{R}} t_2 \xrightarrow{i}_{\mathcal{R}} \cdots \xrightarrow{i}_{\mathcal{R}} t_{m+1} = C[f(w'_1, \ldots, w'_n)]$$

for some $m > 0$. Note that the first rewriting step must be performed at the root of the term $f(\delta_1, \ldots, \delta_n) = t_1$. Here, due to the special shape of the rules in $\mathcal{R}$ (see Proposition 2), we can assume that

1. the sequence performs only one rewriting step using the rule $f(x_1, \ldots, x_n) \to r$ defining f (namely, the first one!) and

2. the occurrence of f in the last term of the sequence has been introduced by the last rule.

This means that there is a cycle $\mathfrak{C}$ consisting of a sequence of pairs $f^\sharp(x_1,\ldots,x_n) \to v_1,\ldots,u_m \to f^\sharp(w_1,\ldots,w_n)$ (where $x_1,\ldots,x_n$ are variables and $w_1,\ldots,w_n$ are terms) and a substitution σ such that $\sigma(f^\sharp(x_1,\ldots,x_n)) = f^\sharp(\delta_1,\ldots,\delta_n)$, $\sigma(v_j) \xrightarrow{i}{}^*_{\mathcal{R}} \sigma(u_{j+1})$ for all j, $1 \le j < m$, and $\sigma(w_j) \xrightarrow{i}{}^*_{\mathcal{R}} \delta'_j$ for all j, $1 \le j \le n$. Furthermore, $\sigma(x_j) = \delta_j$ is a normal form for all variables x_j, $1 \le j \le n$. Hence, by using Proposition 5 together with Definition 2, we can say that $\sigma(v_j) \xrightarrow{i}{}^*_{\mathcal{U}(\mathcal{R},\mathfrak{C})} \sigma(u_{j+1})$ for all j, $1 \le j < m$ and $\sigma(w_j) = \delta'_j$ (hence $\sigma(w_j) \xrightarrow{i}{}^*_{\mathcal{U}(\mathcal{R},\mathfrak{C})} \delta'_j$) for all j, $1 \le j < n$. According to our compatibility assumptions for the rules and dependency pairs with respect to the considered polynomial interpretation, we have that

1. $[\![\sigma(s)]\!] \ge [\![\sigma(t)]\!]$ for all $s \to t \in \mathcal{U}(\mathcal{R},\mathfrak{C}) \cup \mathfrak{C}$ and
2. $[\![\sigma(u)]\!] > [\![\sigma(v)]\!]$ for at least one $u \to v \in \mathfrak{C}$.

In particular, as a consequence of item 1 above, we have $[\![\sigma(w_j)]\!] \ge [\![\delta'_{j+1}]\!]$ for all j, $1 \le j < m$. Furthermore, since our interpretation consists of polynomials with non-negative coefficients only, we have that $[\![f(\ldots,s,\ldots)]\!] \ge [\![f(\ldots,t\ldots)]\!]$ whenever $[\![s]\!] \ge [\![t]\!]$ (weak monotonicity), hence

1. $[\![\sigma(v_j)]\!] \ge [\![\sigma(u_{j+1})]\!]$ for all j, $1 \le j < m$ and
2. $[\![f^\sharp(\sigma(w_1),\ldots,\sigma(w_n))]\!] \ge [\![f^\sharp(\delta',\ldots,\delta'_n)]\!]$.

Since $u_1 = f^\sharp(x_1,\ldots,x_n)$ and $u_i > v_i$ for some $i, 1 \le i \le m$, we conclude

$$\begin{aligned}
[\![f^\sharp(\delta_1,\ldots,\delta_n)]\!] &= [\![\sigma(f^\sharp(x_1,\ldots,x_n))]\!] \\
&> [\![\sigma(f^\sharp(w_1,\ldots,w_n)]\!] \\
&= [\![f^\sharp(\sigma(w_1),\ldots,\sigma(w_n))]\!] \\
&\ge [\![f^\sharp(\delta'_1,\ldots,\delta'_n)]\!]
\end{aligned}$$

Therefore, $[\![f^\sharp(\delta_1,\ldots,\delta_n)]\!] > [\![f^\sharp(\delta'_1,\ldots,\delta'_n)]\!]$.

Since there is no nested cycle in $\mathsf{DG}(\mathcal{R},f^\sharp)$, it follows that all remaining calls to f in the evaluation of t are issued starting from the call $f(\delta'_1,\ldots,\delta'_n)$, i.e.,

1. There is no normal form δ such that $C[\delta] \xrightarrow{i}{}^*_{\mathcal{R}} C'[f(\delta''_1,\ldots,\delta''_n)]$ for normal forms $\delta''_1,\ldots,\delta''_n$ and a context $C'[\,]$, and
2. There is no j, $1 \le j \le n$ such that $\sigma(w_j) \xrightarrow{i}{}^*_{\mathcal{R}} C'[f(w''_1,\ldots,w''_n)]$ for terms $w''_1,\ldots,w''_n$ and context $C'[\,]$.

Thus, $N_f(t) = N_f(f(\delta'_1,\ldots,\delta'_n)) + 1$. By the induction hypothesis, we have $N_f(f(\delta'_1,\ldots,\delta'_n)) \le [\![f^\sharp(\delta'_1,\ldots,\delta'_n)]\!]+1$. Since $[\![f^\sharp(\delta_1,\ldots,\delta_n)]\!] > [\![f^\sharp(\delta',\ldots,\delta'_n)]\!]$, and we use polynomial interpretations over an ordered domain of natural numbers (where $n > m$ is equivalent to $n \ge m+1$), we finally conclude that $N_f(t) \le [\![f^\sharp(\delta_1,\ldots,\delta_n)]\!] + 1$ as desired. □

Theorem 3 *Let $\mathcal{P}$ be a SAFE program, $\mathcal{R} = (\mathcal{F},R)$ be the transformed TRS (i.e., $\mathcal{R} = \mathcal{R}_\mathcal{P}$), and $f \in \mathcal{D}$ be defined in $\mathcal{P}$ and such that $d = \mathsf{ND}(\mathcal{R},f^\sharp) > 0$.*

Let $[\![\ ]\!]$ be a polynomial interpretation over the naturals satisfying that, for all minimal cycles $\mathfrak{C}$ in $\mathsf{DG}(\mathcal{R}, f^\sharp)$, (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, $N_f(t) < (d+1)^{[\![f^\sharp(\delta_1,\ldots,\delta_n)]\!]+1}$.

Proof. If the evaluation of $f(\delta_1, \ldots, \delta_n)$ only involves the cycle $\mathfrak{C}$ whose nesting degree $\mathsf{nd}(\mathfrak{C}, f^\sharp)$ is zero, then we are in the very same case of Theorem 2. Thus, since $d > 0$, we have $N_f(t) \leq [\![f^\sharp(\delta_1, \ldots, \delta_n)]\!] + 1 < (d+1)^{[\![f^\sharp(\delta_1,\ldots,\delta_n)]\!]+1}$ as desired. Therefore, in the remainder of the proof, we assume that nested cycles are involved during the evaluation sequence starting from $f(\delta_1, \ldots, \delta_n)$. Thus, there is an 'outermost' cycle $\mathfrak{C}_1 = \{u_1 \to v_1, \ldots, u_p \to v_p\}$, for $p \geq 1$, such that

1. $u_1 = f^\sharp(x_1, \ldots, x_n)$,
2. $v_p = f^\sharp(v_{p1}, \ldots, v_{pn})$, and
3. there are $0 < k \leq d$ right-hand sides v_i of pairs $u_i \to v_i$ containing occurrences of f, i.e., $v_i = g_i(f(v'_{i1}, \ldots, v'_{in}), v_{i2}, \ldots, v_{im})$.

Note that an outermost cycle automatically 'uses' the innermost ones because there is a call to (an instance of) $f(v'_{i1}, \ldots, v'_{in})$ as part of any rewrite sequence which is represented by cycle $\mathfrak{C}_1$. As in the proof of Theorem 2, we proceed by induction of the number $N_f(t)$ of calls to f in the sequence. If $N_f(t) = 1$, then since all polynomials contain non-negative coefficients only, we have that $[\![f^\sharp(\delta_1, \ldots, \delta_n)]\!] \geq 0$; hence $2^{[\![f^\sharp(\delta_1,\ldots,\delta_n)]\!]+1} \geq 2 > N_f(t)$. If $N_f(t) > 1$, then, we can write the sequence as follows:

$$
\begin{aligned}
t = \underline{f(\delta_1, \ldots, \delta_n)} &\xrightarrow{i}{}^{+}_{\mathcal{R}} g_1(f(\delta_1^1, \ldots, \delta_n^1), w_2^1, \ldots, w_{m_1}^1) \\
&\xrightarrow{i}{}^{+}_{\mathcal{R}} g_1(\delta^1, w_2^1, \ldots, w_m^1) \\
&\xrightarrow{i}{}^{+}_{\mathcal{R}} g_2(f(\delta_1^2, \ldots, \delta_n^2), w_2^2, \ldots, w_{m_2}^2) \\
&\xrightarrow{i}{}^{+}_{\mathcal{R}} g_2(\delta^2, w_2^2, \ldots, w_{m_2}^2) \\
&\xrightarrow{i}{}^{+}_{\mathcal{R}} \cdots \\
&\xrightarrow{i}{}^{+}_{\mathcal{R}} g_k(f(\delta_1^k, \ldots, \delta_n^k), w_2^k, \ldots, w_{m_k}^k) \\
&\xrightarrow{i}{}^{+}_{\mathcal{R}} g_k(\delta^k, w_2^k, \ldots, w_{m_k}^k) \\
&\xrightarrow{i}{}^{+}_{\mathcal{R}} C[f(\delta'_1, \ldots, \delta'_n)] \\
&\xrightarrow{i}{}^{+}_{\mathcal{R}} C[\delta'] \\
&\xrightarrow{i}{}^{*}_{\mathcal{R}} \delta
\end{aligned}
$$

for some context $C[\,]$, terms $w'_1, \ldots, w'_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, and normal forms $\delta'_1, \ldots, \delta'_n$. As in the proof of Theorem 2, we can conclude that:

$$[\![f^\sharp(\delta_1, \ldots, \delta_n)]\!] > [\![f^\sharp(\delta_1^i, \ldots, \delta_n^i)]\!] \text{ for } 1 \leq i \leq k \text{ and } [\![f^\sharp(\delta_1, \ldots, \delta_n)]\!] > [\![f^\sharp(\delta'_1, \ldots, \delta'_n)]\!]$$

because each sequence from $f(\delta_1, \ldots, \delta_n)$ to $g_i(f(\delta_1^i, \ldots, \delta_n^i), w_2^i, \ldots, w_{m_i}^i)$ actually corresponds to a cycle in $\mathsf{DG}(\mathcal{R}, f^\sharp)$ starting from $f^\sharp(x_1, \ldots, x_n)$ and ending in a pair $f^\sharp(u_1^i, \ldots, u_n^i) \to v_i$ such that $\sigma_i(u_j^i) = \delta_j^i$ for some substitution σ_i and all j, $1 \leq j \leq n$.

Since mutually recursive definitions are not allowed in SAFE programs, we know that all calls to f in the evaluation of t are issued starting either from the calls $f^\sharp(\delta_1^i, \ldots, \delta_n^i)$ for $1 \leq i \leq k$ or from $f(\delta'_1, \ldots, \delta'_n)$, i.e., $N_f(t) = N_f(f(\delta'_1, \ldots, \delta'_n)) + \sum_{i=1}^{k} N_f(f(\delta_1^i, \ldots, \delta_n^i)) + 1$. By the induction hypothesis, we have $N_f(f(\delta'_1, \ldots, \delta'_n)) < (d+1)^{[\![f^\sharp(\delta'_1,\ldots,\delta'_n)]\!]+1}$ (i.e., $N_f(f(\delta'_1, \ldots, \delta'_n)) + 1 \leq (d+1)^{[\![f^\sharp(\delta'_1,\ldots,\delta'_n)]\!]+1}$) and $N_f(f(\delta_1^i, \ldots, \delta_n^i)) < (d+1)^{[\![f^\sharp(\delta_1^i,\ldots,\delta_n^i)]\!]+1}$ (equivalently, $N_f(f(\delta_1^i, \ldots, \delta_n^i)) + 1 \leq (d+1)^{[\![f^\sharp(\delta_1^i,\ldots,\delta_n^i)]\!]+1}$). Since $[\![f^\sharp(\delta'_1, \ldots, \delta'_n)]\!] + 1 \leq [\![f^\sharp(\delta_1, \ldots, \delta_n)]\!]$ and $[\![f^\sharp(\delta_1^i, \ldots, \delta_n^i)]\!] + 1 \leq [\![f^\sharp(\delta_1, \ldots, \delta_n)]\!]$, for $1 \leq i \leq k$, it follows that $N_f(f(\delta'_1, \ldots, \delta'_n)) + 1 \leq (d+1)^{[\![f^\sharp(\delta_1,\ldots,\delta_n)]\!]}$ and $N_f(f(\delta_1^i, \ldots, \delta_n^i)) + 1 \leq (d+1)^{[\![f^\sharp(\delta_1,\ldots,\delta_n)]\!]}$, for $1 \leq i \leq k$. Therefore, since $k \leq d$,

$$\begin{aligned} N_f(t) &= N_f(f(\delta'_1, \ldots, \delta'_n)) + \textstyle\sum_{i=1}^{k} N_f(f(\delta_1^i, \ldots, \delta_n^i)) + 1 \\ &\leq N_f(f(\delta'_1, \ldots, \delta'_n)) + \textstyle\sum_{i=1}^{d} N_f(f(\delta_1^i, \ldots, \delta_n^i)) + 1 \\ &< (d+1) \cdot (d+1)^{[\![f^\sharp(\delta_1,\ldots,\delta_n)]\!]} \\ &= (d+1)^{[\![f^\sharp(\delta_1,\ldots,\delta_n)]\!]+1} \end{aligned}$$

□