Certification of Safe Polynomial Memory Bounds ⋆
(Extended Version)

Javier de Dios and Ricardo Peña

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
jdcastro@aventia.com, ricardo@sip.ucm.es

Abstract. In previous works, we have developed several algorithms for inferring upper bounds to heap and stack consumption for a simple functional language called *Safe*. The bounds inferred for a particular recursive function with n arguments takes the form of symbolic n-ary functions from $(\mathbb{R}^+)^n$ to $\mathbb{R}^+$ relating the input argument sizes to the number of cells or words respectively consumed in the heap and in the stack. Most frequently, these functions are multivariate polynomials of any degree, although exponential and other functions can be inferred in some cases. Certifying memory bounds is important because the analyses could be unsound or have been wrongly implemented. But the certifying process should not be necessarily tied to the method used to infer those bounds. Although the motivation for the work presented here is certifying the bounds inferred by our compiler, we have developed a certifying method which could equally be applied to bounds computed by hand.
The certification process is divided into two parts: (a) an off-line part consisting of proving the soundness of a set of proof rules. This part is independent of the program being certified, and its correctness is established once forever by using the proof assistant Isabelle/HOL; and (b) a compile-time program-specific part in which the proof rules are applied to a particular program and their premises proved correct.
The key idea for the first part is proving an Isabelle/HOL theorem for each syntactic construction of the language, relating the symbolic information asserted by the proof-rule to the dynamic properties about the heap and stack consumption satisfied at runtime. For the second part, we use a mathematical tool for proving instances of the Tarski decision problem on quantified formulas in real closed fields.
keywords: Memory bounds, formal certificates, proof assistants, Tarski decision problem.

1 Introduction

Certifying program properties consists of providing mathematical evidence about them. In a Proof Carrying Code (PCC) environment [17], these proofs should

⋆ Work supported by the projects TIN2008-06622-C03-01/TIN (STAMP) and S2009/TIC-1465 (PROMETIDOS).

be checked by an appropriate tool. The certified properties may be obtained either manually, interactively, or automatically, but whatever is the effort needed for generating them, the PCC paradigm insists on their checking to be fully automatic.

In our setting, the certified property (safe memory bounds) is automatically inferred as the product of several static analyses, so that the certificate can be generated by the compiler without any human intervention. Certifying the inferred property is needed to convince a potential consumer that the static analyses are sound and that they have been correctly implemented in the compiler.

Inferring safe memory bounds in an automatic way is a complex task, involving in our case several static analyses:

- A region inference anaysis [15] decides in which regions different data structures should be allocated, so that they could be safely destroyed when the region is deallocated. At the same time, the live memory is kept to a minimum (in other words, the analysis detects the maximum possible garbage).
- A size analysis infers upper bounds to the size of certain variables.
- A termination analysis [14] is used to infer upper bounds to the number of internal calls of recursive functions.
- A space inference analysis [16], uses the results of the above analyses to infer upper bounds to the heap and stack consumption.

Memory bounds could also be manually obtained, but in this case the computation must determine all the additive and multiplicative constants. This is usually a tedious and error-prone task.

But, once the memory bounds have been obtained, certifying them should be a simpler task. It is common folklore in the PCC framework that to find a proof is always more complex than to check it. A good example of this is ranking function synthesis in termination proofs of recursive and iterative programs. A ranking function is a kind of certificate or *witness* of termination. To find them is a rather complex task. Sometimes, linear methods [20] or sophisticated polyhedra libraries are used [10, 1]. Others, more powerful methods such as SAT solvers [3] or non-linear constraint solvers [11] are needed. But, once the ranking function has been obtained, certifying termination consists of 'simply'[1] proving that it strictly decreases at each program transition in some well-founded order. This shows that the certifying and the inference processes are not necessarily tied.

In this paper we propose a simple way of certifying upper memory bounds whatever complex the method to obtain them has been. In the first part, we develop a set of syntax-driven proof-rules allowing to infer safe upper memory bounds to the execution of any expression, provided we have already upper bounds for its subexpressions. Then we prove their soundness by relating the symbolic information inferred by a rule to the dynamic properties about the heap and stack consumption satisfied at runtime. In order to get complete confidence on the rules, we have used the Isabelle/HOL proof assistant [19] for this task.

[1] If the ranking function is not linear, this checking may not be so simple, and even it might be undecidable.

```
unshuffle []     = ([],[])
unshuffle (x:xs) = (x:ys2, ys1)
                   where (ys1,ys2) = unshuffle xs

merge []     ys  = ys
merge (x:xs) []  = x:xs
merge (x:xs) (y:ys) | x <= y    = x : merge xs     (y:ys)
                    | otherwise = y : merge (x:xs) ys

msort []  = []
msort [x] = [x]
msort xs  = merge (msort xs1) (msort xs2)
            where (xs1, xs2) = unshuffle xs
```

Fig. 1. *mergesort* algorithm in *Full-Safe*

In the second part we explain how, given a candidate upper bound for a recursive function, the compiler can apply the proof-rules and infer a new upper bound, which will be correct provided the candidate upper bound is correct. Our main proof-rule states that if the derived bound *is smaller than or equal to the candidate one*, then both are correct. In order to certify this latter inequality, we propose to use a computer algebra tool for proving instances of Tarski's decision problem on quantified formulas involving polynomials over the reals [21]. To our knowledge, this is the first time that the described method is used to certify memory upper bounds.

The plan of the paper is as follows: after this introduction, in Sec. 2 we briefly summarize the characteristics and semantics of our functional language *Safe*; sections 3, 4, and 5 are devoted to presenting the proof-rules and to proving their soundness; Sec. 6 explains the certification process and how a symbolic algebra tool is used as a certificate checker; Sec. 7 presents a small case study illustrating the certificate generation and checking; finally, Sec. 8 presents some related work and draws the paper conclusions.

2 The language

Safe is a first-order eager language with a syntax similar to Haskell's. Fig. 1 shows a mergesort algorithm written in *Full-Safe*. Its runtime system uses *regions*, i.e. disjoint parts of the heap where the program allocates data structures. They are automatically inferred [15] and made explicit in the intermediate language, called *Core-Safe*, and in the internal types. For instance, the types inferred for the functions of Fig. 1 are (the ρ's are region types):

$$\begin{aligned} \mathit{unshuffle} &:: [a]@\rho \to \rho_1 \to \rho_2 \to ([a]@\rho_1, [a]@\rho_1)@\rho_2 \\ \mathit{merge} &:: [a]@\rho \to [a]@\rho \to \rho \to [a]@\rho \\ \mathit{msort} &:: [a]@\rho' \to \rho \to [a]@\rho \end{aligned}$$

The smallest memory unit is the *cell*, a contiguous memory space big enough to hold a data construction. A cell contains the mark of the constructor and a representation of the free variables to which the constructor is applied. These may consist either of basic values, or of pointers to other constructions. Each cell is allocated at constructor application time. A *region* is a collection of cells. It is

$$\frac{}{E \vdash (h,k), td, c \Downarrow (h,k), c, ([\,]_k, 0, 1)}\ [Lit]$$

$$\frac{}{E[x \mapsto v] \vdash (h,k), td, x \Downarrow (h,k), v, ([\,]_k, 0, 1)}\ [Var]$$

$$\frac{j \le k \quad fresh(p)}{E \vdash (h,k), td, C\ \overline{a_i}^n\ @\ r \Downarrow (h \uplus [p \mapsto (j, C\ \overline{v_i}^n)], k), p, ([j \mapsto 1]_k, 1, 1)}\ [Cons]$$

$$\frac{(f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e) \in \Sigma_D \quad [\overline{x_i \mapsto E(a_i)}^n, \overline{r_j \mapsto E(r'_j)}^l, self \mapsto k+1] \vdash (h, k+1), n+l, e \Downarrow (h', k+1), v, (\delta, m, s)}{E \vdash (h,k), td, f\ \overline{a_i}^n\ @\ \overline{r'_j}^l \Downarrow (h'|_k, k), v, (\delta|_k, m, \max\{n+l, s+n+l-td\})}\ [App]$$

$$\frac{E \vdash (h,k), td, a_1 \Downarrow (h,k), v_1, ([\,]_k, 0, 1) \quad E \vdash (h,k), td, a_2 \Downarrow (h,k), v_2, ([\,]_k, 0, 1)}{E \vdash (h,k), td, a_1 \oplus a_2 \Downarrow (h,k), v_1 \oplus v_2, ([\,]_k, 0, 2)}\ [Primop]$$

$$\frac{\begin{array}{c} E \vdash (h,k), 0, e_1 \Downarrow (h',k), v_1, (\delta_1, m_1, s_1) \\ E \cup [x_1 \mapsto v_1] \vdash (h',k), td+1, e_2 \Downarrow (h'',k), v, (\delta_2, m_2, s_2) \end{array}}{E \vdash (h,k), td, \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 \Downarrow (h'',k), v, (\delta_1 + \delta_2, \max\{m_1, |\delta_1| + m_2\}, \max\{2 + s_1, 1 + s_2\})}\ [Let]$$

$$\frac{C = C_r \quad E \cup [\overline{x_{r_i} \mapsto v_i}^{n_r}] \vdash (h,k), td + n_r, e_r \Downarrow (h',k), v, (\delta, m, s)}{E[x \mapsto p] \vdash (h[p \mapsto (j, C\ \overline{v_i}^n)], k), td, \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \Downarrow (h',k), v, (\delta, m, s + n_r)}\ [Case]$$

Fig. 2. Resource-Aware Operational semantics of *Core-Safe* expressions

created empty and it may grow up while it is active. Region deallocation frees all its cells. The allocation and deallocation of regions are bound to function calls. A *working region*, denoted by *self*, is allocated when entering the call and deallocated when exiting it. Inside the function, data structures not belonging to the output may be built there.

Fig. 2 shows the *Core-Safe* big-step semantic rules, with extra annotations added in order to obtain the resources used by evaluating an expression. A judgment of the form $E \vdash (h,k), td, e \Downarrow (h',k), v, (\delta, m, s)$ means that expression e is evaluated in an environment E using the td topmost stack positions, and in a heap (h,k) with $0, \ldots, k$ active regions. As a result, a heap (h',k) and a value v are obtained, and a resource vector (δ, m, s), explained below, is consumed.

We denote data constructors by C, constants by c, variables by x, and atoms —an atom is either a constant or a variable— by a. Σ_D is a global environment containing all the function definitions. By $h|_k$ we denote the heap h with all regions above k deleted. A heap h is a mapping between pointers p and constructor cells $(j, C\ \overline{v_i}^n)$, where j is the cell region. The first component of the resource vector is a partial function $\delta : \mathbb{N} \to \mathbb{N}$ giving for each active region i the difference between the cells in the final and initial heaps. By $dom\ \delta$ we denote the subset $\{0 \ldots k\}$ in which δ is defined. By $[\,]_k$ we denote the function $\lambda i \in \{0 \ldots k\}.0$. By $|\delta|$ we mean the sum $\sum_{i \in dom\ \delta} \delta\ i$ giving the total balance of cells. The remaining components m and s respectively give the *minimum* number of fresh cells in the heap and of words in the stack needed to successfully evaluate e, i.e. the peak memory used during e's evaluation. When e is the main expression, these figures give us the total memory needs of a particular run of the *Safe* program.

3 Function Signatures

A *Core-Safe* function is defined as a $n + m$ argument expression

$$f :: t_1 \to \ldots \to t_n \to \rho_1 \to \ldots \to \rho_m \to t$$
$$f\ x_1 \cdots x_n\ @\ r_1\ \cdots r_m = e_f$$

where $r_1\ \cdots r_m$ are the region arguments. A function may charge space costs to heap regions and to the stack. In general, these costs depend on the *sizes* of the function arguments. We define the size of an algebraic type term to be the number of cells of its recursive spine. This is always at least 1. We define the size of a boolean value to be zero. However, for an integer argument we choose its size to be its value because frequently space costs depend on the value of a numeric argument. As a consequence, all the costs and sizes of a function f can be expressed as functions on f's argument sizes:

$$\mathbb{F}_f = \{\eta : (\mathbb{R}^+ \cup \{+\infty\})^n \to \mathbb{R}^+ \cup \{+\infty, -\infty\} \ \mid\ \eta \text{ is monotonic}\}$$

Cost or size $+\infty$ are used to represent that the analysis is not able to infer a bound, while $-\infty$ is used to express that the cost or size is not defined. For instance, the following function, where xs is assumed to be a list size,

$$\lambda xs. \begin{cases} xs - 3 & \text{if } xs \geq 4 \\ -\infty & \text{otherwise} \end{cases}$$

is undefined for sizes xs smaller than 4 (i.e. for lists with less than 3 elements).

They are ordered as expected, $-\infty \leq 0$, and $\forall x \in \mathbb{R}^+.x \leq +\infty$, so $-\infty \sqcup x = x$ and $+\infty \sqcup x = +\infty$. Arithmetic monotonic operations with $\pm\infty$ are defined as follows, where $x \in \mathbb{R}^+$ while $y \in \mathbb{R}^+ \cup \{+\infty, -\infty\}$:

$$-\infty + y = -\infty \quad -\infty * y = -\infty \quad +\infty + x = +\infty \quad +\infty * x = +\infty$$

The domain of cost functions $(\mathbb{F}_f, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a complete lattice with the usual order $\sqsubseteq$ between functions. The rest of the components are standard. Notice that it is closed under the operations $\{+, \sqcup, *\}$.

Function f above may charge space costs to a maximum of $m + 1$ regions: it may create cells in any output region $r_1 \ldots r_m$, and additionally in its *self* region. Each region r has a region type. We denote by R_f the set $\{\rho_1 \ldots \rho_m\}$ of argument region types, and by ρ^f_{self} the type of region *self*.

Looked from outside, the charges to the *self* region are not visible, so $\mathbb{D}_f = \{\Delta : R_f \to \mathbb{F}_f\}$ is the complete lattice of functions describing the space costs charged by f to visible regions. We will call *abstract heaps* to these functions.

Definition 1. *A function* signature *for f is a triple $(\Delta_f, \mu_f, \sigma_f)$, where Δ_f belongs to $\mathbb{D}_f$, and μ_f, σ_f belong to $\mathbb{F}_f$.*

The aim is that Δ_f is an upper bound to the cost charged by f to visible regions, (i.e. to the increment in *live memory* due to a call to f), and μ_f, σ_f respectively are upper bounds to the heap and stack *peaks* contributed by f's evaluation.

$$\theta, \phi, td \rhd_f c, \Sigma \vdash ([\,]_f, 0, 1) \quad [Lit]$$

$$\theta, \phi, td \rhd_f x, \Sigma \vdash ([\,]_f, 0, 1) \quad [Var]$$

$$\theta, \phi, td \rhd_f a_1 \oplus a_2, \Sigma \vdash ([\,]_f, 0, 2) \quad [Primop]$$

$$\theta, \phi, td \rhd_f C\ \overline{a_i}^n\ @\ r, \Sigma \vdash ([\theta\ r \mapsto 1]_f, 1, 1) \quad [Cons]$$

$$\frac{\begin{array}{lll}\Sigma\ g = (\Delta_g, \mu_g, \sigma_g) & G\ (\overline{a_i}^n) \equiv \bigwedge_{i=1}^{l} def(\phi\ a_i\ \overline{x}^n) & argP(\psi, \overline{\rho_j}^q, \theta, \overline{r_j}^q) \\ \mu = \lambda \overline{x}^n.[G\ (\overline{a_i}^n) \rightarrow \mu_g\ (\overline{\phi\ a_i\ \overline{x}^n}^l)] & \sigma = \lambda \overline{x}^n.[G\ (\overline{a_i}^n) \rightarrow \sigma_g\ (\overline{\phi\ a_i\ \overline{x}^n}^l)] & \Delta = instance_f(\Delta_g, \psi, \overline{a_i}^l)\end{array}}{\theta, \phi, td \rhd_f g\ \overline{a_i}^l\ @\ \overline{r_j}^q, \Sigma \vdash (\Delta, \mu, \sqcup\{l + q, \sigma + l + q - td\})} \quad [App]$$

$$\frac{\theta, \phi, 0 \rhd_f e_1, \Sigma \vdash (\Delta_1, \mu_1, \sigma_1) \quad \theta, \phi, td + 1 \rhd_f e_2, \Sigma \vdash (\Delta_2, \mu_2, \sigma_2)}{\theta, \phi, td \rhd_f \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, \Sigma \vdash (\Delta_1 + \Delta_2, \sqcup\{\mu_1, |\Delta_1| + \mu_2\}, \sqcup\{2 + \sigma_1, 1 + \sigma_2\})} \quad [Let]$$

$$\frac{(\forall i)\ \theta, \phi, td + n_i \rhd_f e_i, \Sigma \vdash (\Delta_i, \mu_i, \sigma_i)}{\theta, \phi, td \rhd_f \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^n, \Sigma \vdash (\bigsqcup_{i=1}^n \Delta_i, \bigsqcup_{i=1}^n \mu_i, \bigsqcup_{i=1}^n (\sigma_i + n_i))} \quad [Case]$$

Fig. 3. Proof-rules for *Core-Safe* expressions

$$\frac{(f\ \overline{x_i}^l @\ \overline{r_j}^q = e_f) \in \Sigma_D \quad \theta, \phi, l + q \rhd_f e_f, \Sigma \uplus \{f \mapsto (\Delta, \mu, \sigma)\} \vdash (\Delta', \mu', \sigma') \quad (\lfloor \Delta' \rfloor, \mu', \sigma') \sqsubseteq (\Delta, \mu, \sigma)}{\theta, \phi, l + q \rhd_f e_f, \Sigma \vdash (\Delta', \mu', \sigma')} \quad [Rec]$$

Fig. 4. Proof-rule for a (possibly) recursive *Core-Safe* function definition

4 Proof-rules

When dealing with an expression e, we assume it belongs to the body e_f of a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$, that we will call the *context function*, assumed to be well-typed.

We consider available a local type environment θ giving the types of all (free and bound) variables in e_f. It allows to type e_f and all its subexpressions. We also consider available a local environment ϕ giving for every (free and bound) variable its size as a symbolic function of the sizes of f's formal arguments $\overline{x_i}^n$. Let Σ be a global environment giving, for each *Safe* function g in scope, its signature $(\Delta_g, \mu_g, \sigma_g)$, and let td (abbreviation of *top-depth*) be a natural number. This is a quantity used by the compiler to control the size of the runtime environment stored in the stack (it is the same argument used in the operational semantics, see Sec. 2). It has an impact on the stack consumption and so it will be needed in our judgements. Finally, we consider available as an implicit global constant, a type environment Σ_T giving for every function and data constructor of the program their most general types.

We inductively define a *derivation* relation as a set of proof-rules. The intended meaning of a judgement of the form $\theta, \phi, td \rhd_f e, \Sigma \vdash (\Delta, \mu, \sigma)$ is that Δ, μ, σ are safe upper bounds for respectively the live heap contributed by evaluating the expression e, the additional peak heap needed by e, and its additional peak stack. The context information needed is: a valid global signature environment Σ, two valid local environments θ (for types) and ϕ (for sizes), a runtime enviroment top depth td, and the name f of the context function.

In Figure 3 we show the proof-rules for the most relevant *Core-Safe* expressions. Predicate $def(\eta)$ expresses that the size η is defined according to its type: if η has an algebraic type, $def(\eta) \equiv \eta \geq 1$; if it is an integer, $def(\eta) \equiv \eta \geq 0$; otherwise $def(\eta) \equiv True$. We use the guarded notation $[G \rightarrow \eta]$, as equivalent to η if G holds, and to $-\infty$ otherwise. By $[\,]_f$ we denote the constant function $\lambda\rho \in R_f \cup \{\rho^f_{self}\} . \lambda\overline{x_i}^n . [def(\overline{x_i}^n) \rightarrow 0]$, and by $[\rho' \rightarrow \eta]_f$ we denote the function:

$$\lambda\rho \in R_f \cup \{\rho^f_{self}\} \,.\, \lambda\overline{x_i}^n \,.\, \begin{cases} [def(\overline{x_i}^n) \rightarrow 0] & \text{if } \rho \neq \rho' \\ [def(\overline{x_i}^n) \rightarrow \eta] & \text{if } \rho = \rho' \end{cases}$$

We abbreviate $\lambda\overline{x_i}^n \,.\, [def(\overline{x_i}^n) \rightarrow c]$ by c, when $c \in \mathbb{R}^+$. By $|\Delta|$ we mean $\sum_{\rho \in dom\ \Delta} \Delta\ \rho$.

Rules $[Lit]$, $[Var]$, $[Primop]$ and $[Cons]$ exactly reflect the corresponding resource-aware semantic rules shown in Fig. 2.

When a function application $g\ \overline{a_i}^l\ @\ \overline{r_j}^q$ is found, its signature $\Sigma\ g$ is applied to the sizes of the actual arguments, $\overline{\phi\ a_i\ \overline{x_j}^n}^l$. Some different region types of g may instantiate to the same actual region type of f. This instantiation mapping $\psi : R_g \rightarrow R_f \cup \{\rho^f_{self}\}$ is provided by the compiler, and we will require it to be consistent with the typing environment θ and with the actual region arguments of the application. We call this property *to be argument preserving*.

Definition 2. *Given a type signature for g, $\Sigma_T\ g = \overline{t_i}^l \rightarrow \overline{\rho_j}^q \rightarrow t$, a local type environment θ for f, and the sequence of actual region arguments $\overline{r_j}^q$, we say that the instantiation mapping ψ is* argument preserving *with respect to $\overline{\rho_j}^q$, θ, and $\overline{r_j}^q$, denoted $argP(\psi, \overline{\rho_j}^q, \theta, \overline{r_j}^q)$, if $\forall j \in \{1 \ldots q\} \,.\, \psi\ \rho_j = \theta\ r_j$.*

The memory consumed by g in the formal regions mapped by ψ to the same f's actual region must be accumulated in order to get the charge to this region of f. In the $[App]$ rule of Figure 3, function $instance_f$ does this computation.

Definition 3. *The funcion $instance_f$ converts an abstract heap for g into an abstract heap for f. If f is the context function, $\Sigma\ g = (\Delta_g, \_, \_)$, ϕ is the local size environment for f, ψ is an instantiation mapping, and $\overline{a_i}^l$ are the arguments of the application, we define $instance_f(\Delta_g, \psi, \overline{a_i}^l)$ as the abstract heap Δ with domain $R_f \cup \{\rho^f_{self}\}$ such that:*

$$\forall \rho \in dom\ \Delta \,.\, \Delta\ \rho = \lambda\ \overline{x_i}^n \,.\, [G\ (\overline{a_i}^n) \rightarrow \sum_{\rho' \in R_g \wedge \psi\ \rho' = \rho} \Delta_g\ \rho'\ (\overline{\phi\ a_i\ \overline{x_i}^n}^l)]$$

where $G\ (\overline{a_i}^n) \equiv \bigwedge_{i=1}^{l} def(\phi\ a_i\ \overline{x}^n)$. Notice that if any of the sizes $\phi\ a_i\ \overline{x_i}^n$ is not defined, Δ_g applied to it is neither defined. It is easy to see that Δ, μ and σ defined in rule $[App]$ are monotonic. If $\exists i \in \{1 \ldots l\} \,.\, \neg def(\phi\ a_i\ \overline{x_i}^n)$, then Δ, μ and σ return $-\infty$, which guarantees monotonicity since $-\infty$ is the smallest value in the domain. For the rest of the arguments, monotonicity is guaranteed by the monotonicity of Δ_g, μ_g and σ_g.

Rule $[Let]$ reflects the corresponding resource-aware semantic rule, while rule $[Case]$ uses the least upper bound operators $\bigsqcup$ in order to obtain an upper bound to the costs and of all the branches.

In Fig. 4 we show the proof rule for recursive functions. In fact, it could also be applied to non-recursive ones. By $\lfloor \Delta \rfloor$ we denote the projection of Δ over R_f, obtained by removing the region ρ^f_{self} from Δ. This rule is a relevant contribution of the paper, since it reduces proving upper memory bounds to checking inequalities between functions over the reals. In words, its says that if a triple (Δ, μ, σ) (obtained by whatever means) is to be proved a safe upper bound for the recursive function f, a sufficient condition is:

1. Introduce (Δ, μ, σ) in the environment Σ as a candidate signature for f.
2. By using the remaining proof-rules, derive a triple (Δ', μ', σ') as a new upper bound for f's body.
3. Prove $(\lfloor \Delta' \rfloor, \mu', \sigma') \sqsubseteq (\Delta, \mu, \sigma)$.

The rule asserts that (Δ', μ', σ') is a correct bound for e_f without any assumption for f in Σ. By deleting the *self* region, then $(\lfloor \Delta' \rfloor, \mu', \sigma')$ is a correct signature for f, and so will it be (Δ, μ, σ), which is greater than or equal to it.

The first two steps are rutinary. The only difficulty remaining is proving the third. As we will see, for polynomial functions this can be done by converting it into a decision problem of Tarski's theory of closed real fields.

5 Soundness theorems

Let $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$, be the context function and θ the inferred local type environment for f. Let us assume an execution of e_f under some $h_0, k_0 = k+1$, and $E_0 = [\overline{x_i \mapsto v_i}^n, \overline{r_j \mapsto i_j}^m, self \mapsto k+1]$ for some v_i, $0 \leq i_j \leq k$ for all j, and $td_0 = n + m$:

$$E_0 \vdash h_0, k_0, td_0, e_f \Downarrow h_f, k_0, v_f, (\delta_0, m_0, s_0) \tag{1}$$

In the following, all $\Downarrow$–judgements corresponding to a given sub-expression of e_f will be assumed to belong to the derivation of (1). Also, we will call generically a *bound* to a triple (Δ, μ, σ).

The steps we shall follow in this section are: (1) we shall introduce a notion of semantic satisfaction of a bound by an expression in the context of a given function; (2) we shall define a notion of valid signature which formalises the intuition of a bound $(\Delta_g, \mu_g, \sigma_g)$ actually being an upper bound of the actual (δ, m, s) obtained in any execution of the function g. This will lead us to the notion of *valid global bound environment* Σ; (3) we shall refine the notion of semantic satisfaction of a bound to a conditional one subject to the validity of a global bound environment; and (4) we shall prove that the proof rules of figures 3 and 4 are sound with respect to the given semantic notions.

We need to provide the following definitions:

1. A notion of size of a data structure, so that we can refer to the sizes of the arguments of a function.
2. A notion of valid type and size local environments, whose properties will be needed when proving the soundness of the proof rules.

Definition 4. *Given a value v (either a constant c or a pointer p) belonging to a heap h, the function size returns the number of cells in h of the data structure starting at v:*

$$\begin{array}{ll} size(h, c) & = 0 \\ size(h[p \mapsto (j, C\ \overline{v_i}^n)], p) & = 1 + \sum_{i \in RecPos\ C} size(h, v_i) \end{array}$$

where RecPos C denotes the recursive positions of constructor C, and they are obtained by consulting the type $\Sigma_T\ C$.

For example, if p points to the first cons cell of the list $[1, 2, 3]$ in the heap h then $size(h, p) = 4$.

When applying abstract signatures to sizes, as in Definition 8 below, we will consider that the size of an actual integer argument is its value, i.e. $size(h, n) = n$ for every numeric argument n. This is so, because very frequently the memory consumption of a function depends on the value of its integer arguments.

We reproduce from previous papers the notions of a region runtime instantiation mapping η to be *admissible*, and of consistency between the static types and the runtime contents of the heap.

Definition 5. *Assuming that k denotes the topmost region of a given heap, we say that the mapping η is* admissible*, denoted admissible $\eta\ k$, if:*

$$\rho^f_{self} \in dom\ \eta \wedge \eta\ \rho^f_{self} = k \wedge \forall \rho \in (dom\ \eta) - \{\rho^f_{self}\}\ .\ \eta\ \rho\ < k$$

Definition 6. *We say that the mappings θ, η, the runtime environment E, and the heap h are* consistent*, denoted consistent $\theta\ \eta\ E\ h$, if:*

1. $\forall x \in dom(E)\ .\ consistent(\theta(x), \eta, E(x), h)$ *where:*

$$\begin{array}{lll} consistent(B, \eta, c, h) & = true & \text{-- } B \text{ denotes a basic type} \\ consistent(a, \eta, v, h) & = true & \text{-- } a \text{ denotes a type variable} \\ consistent(t, \eta, p, h) & \Leftarrow p \notin dom(h) & \\ consistent(T\ \overline{t'_i}^m @\ \overline{\rho_j}^l, \eta, p, h) & \Leftarrow \exists j\ C\ \overline{v_k}^n\ \mu\ \overline{t_{kC}}^n\ \overline{\rho_{jC}}^l\ .\ h(p) = (j, C\ \overline{v_k}^n) & \\ & \wedge\ \rho_l \in dom(\eta)\ \wedge\ \eta(\rho_l) = j & \\ & \wedge\ \Gamma_T(C) = \overline{t_{kC}}^n \to \rho_{lC} \to T\ \overline{t'_{iC}}^m @\ \overline{\rho_{jC}}^l & \\ & \wedge\ \mu(T\ \overline{t'_{iC}}^m @\ \overline{\rho_{jC}}^l) = T\ \overline{t'_i}^m @\ \overline{\rho_j}^l & \\ & \wedge\ \forall k \in \{1..n\}\ .\ consistent(\mu(t_{kC}), \eta, v_k, h)) & \end{array}$$

2. $\forall r \in dom\ E\ .\ \theta\ r \in dom\ \eta\ \wedge\ E\ r = (\eta \cdot \theta)\ r$
3. $self \in dom\ E \wedge \theta\ self = \rho^f_{self}$

Now me move to the validity of local type and size environments.

Definition 7 (Validity of local environments). *Let $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$ be the context function, e a subexpression of e_f, and θ, ϕ respectively be local type and size environments for f. We say that they are* valid*, denoted valid$_f$ $\theta\ \phi$, if*

$$\begin{array}{l} (\overline{x_i}^n \cup fv\ e \cup \overline{r_j}^m \cup self) \subseteq dom\ \theta \\ \wedge\ (\overline{x_i}^n \cup fv\ e) \subseteq dom\ \phi \\ \wedge\ (\forall E\ h\ k\ td\ h'\ v\ \eta\ \overline{s_i}\ \delta\ m\ s\ . \\ \quad E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s) \wedge (\forall i\ .\ s_i = size(h, E\ x_i)) \wedge admissible\ \eta\ k \to \\ \quad consistent\ \theta\ \eta\ E\ h \wedge (\forall y \in dom\ \phi\ .\ \phi\ y\ \overline{s_i}^n \geq size(h, E\ y))) \end{array}$$

The semantic satisfaction of a bound by an expression, denoted

$$\theta, \phi, td \rhd_f e \models [\![(\Delta, \mu, \sigma)]\!]$$

must express that, whenever θ, ϕ are valid environments, and some minor static and dynamic properties hold, then (Δ, μ, σ) is a correct bound for the memory consumption of expression e in any of its possible evaluations.

Definition 8. *Let $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$ be the context function, and e a subexpression of e_f. We say that e satisfies the bound (Δ, μ, σ) in the context of θ, ϕ, denoted $\theta, \phi, td \rhd_f e \models [\![(\Delta, \mu, \sigma)]\!]$, if:*

$$valid_f\ \theta\ \phi \rightarrow$$
$$P_{stat} \wedge (\forall E\ h\ k\ h'\ v\ \eta\ \delta\ m\ s\ \overline{s_i}^n\ .\ P_{\Downarrow} \wedge P_{dyn} \wedge P_{size} \wedge P_{\eta} \rightarrow P_{\Delta} \wedge P_{\mu} \wedge P_{\sigma})$$

where:

1. $P_{stat} \stackrel{def}{=} dom\ \Delta = R_f \cup \{\rho^f_{self}\}$
2. $P_{\Downarrow} \stackrel{def}{=} E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$
3. $P_{dyn} \stackrel{def}{=} (\overline{x_i}^n \cup fv\ e \cup \overline{r_j}^m \cup self) \subseteq dom\ E \wedge dom\ \eta = dom\ \Delta$
4. $P_{size} \stackrel{def}{=} \forall i \in \{1..n\}\ .\ s_i = size(h, E\ x_i)$
5. $P_{\eta} \stackrel{def}{=} admissible(\eta, k)$
6. $P_{\Delta} \stackrel{def}{=} \forall j \in \{0 \ldots k\}\ .\ \sum_{\eta\ \rho=j}\ \Delta\ \rho\ \overline{s_i}^n \geq \delta\ j$
7. $P_{\mu} \stackrel{def}{=} \mu\ \overline{s_i}^n \geq m$
8. $P_{\sigma} \stackrel{def}{=} \sigma\ \overline{s_i}^n \geq s$

The semantic satisfaction of a bound by a whole function's body will convert this bound in a *bound signature* for the function, These signatures can be stored in a global bound environment which we will call a *valid* environment.

Definition 9. *A global bound environment Σ is* valid*, denoted $\models \Sigma$, if it belongs to the following inductively defined set:*

1. $\models \emptyset$*, i.e. the empty environment is always valid.*
2. *If $\models \Sigma$, and $f\ \overline{x_i}^l\ @\ \overline{r_j}^m = e_f$, and there exist Δ, μ, σ and valid local environments θ, ϕ such that $\theta, \phi, (l+m) \rhd_f e_f \models [\![(\Delta, \mu, \sigma)]\!]$, then $\models \Sigma \uplus \{f \mapsto (\lfloor \Delta \rfloor, \mu, \sigma)\}$.*

When proving a bound for an expression, we will usually need a valid global environment in order to get from it correct signatures for the subsidiary functions called by the expression, and then being able to apply the *App* proof rule. We will then say that the satisfaction of the bound is *conditioned* to the validity of the environment.

Definition 10. *We say that an expression e* conditionally satisfies *a bound (Δ, μ, σ) with respect to a bound environment Σ in the context of θ, ϕ, denoted $\theta, \phi, td \rhd_f e, \Sigma \models [\![(\Delta, \mu, \sigma)]\!]$, if $\models \Sigma \rightarrow \theta, \phi, td \rhd_f e \models [\![(\Delta, \mu, \sigma)]\!]$.*

Now, we are in a position to state and prove the main theorem establishing that the proof rules of figures 3 and 4 are sound.

Theorem 1 (Soundness).

$$\textit{If}\ \ \theta, \phi, td \rhd_f e, \Sigma \vdash (\Delta, \mu, \sigma),\ \ \textit{then}\ \ \theta, \phi, td \rhd_f e, \Sigma \models [\![(\Delta, \mu, \sigma)]\!]$$

5.1 Proving the soundness theorem

It will take us some work to set up the infrastructure for proving the theorem. First, we enrich the big-step semantics with a side-effect counter n_f counting the *maximum length* of the recursive call chains to a given function f. Then, $E \vdash h, k, td, e \Downarrow_f h', k, v, (\delta, m, s), n_f$ means that, in the derivation tree of expression e, there are 0 or more calls to f, and the longest call-chain involving f and starting at e has a length $n_f \geq 0$. A length $n_f = 0$ means that there are no calls to f during e's evaluation.

Definition 11. *We define a restricted big-step semantics with an upper bound n to the longest chain of f's:*

$$E \vdash h, k, td, e \Downarrow_{f,n} h', k, v, (\delta, m, s) \stackrel{def}{=} E \vdash h, k, td, e \Downarrow_f h', k, v, (\delta, m, s), n_f \wedge n_f \leq n$$

If we write $P_{\Downarrow}(f, n)$ we refer to the following property:

$$E \vdash h, k, td, e \Downarrow_{f,n} h', k, v, (\delta, m, s)$$

which is similar to $P_{\Downarrow}$ of Def. 8 but using the $\Downarrow_{f,n}$ relation instead of the $\Downarrow$ one. The following lemma establishes that both semantics are in essence equivalent.

Lemma 1. *$E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$ if and only if*

$$\exists n \,.\, E \vdash h, k, td, e \Downarrow_{f,n} h', k, v, (\delta, m, s)$$

Proof. By induction on $\Downarrow_{f,n}$ the *if* direction, and by induction on $\Downarrow$ the *only if* one.

We modify definitions 8, 9, and 10 in order to introduce the depth of the derivation:

Definition 12. *Let $f\ \overline{x_i}^l\ @\ \overline{r_j}^m = e_f$ be the context function, and e a subexpression of e_f. We say that e satisfies the bound (Δ, μ, σ) in the context of θ, ϕ, up to depth n for f, denoted $\theta, \phi, td \rhd_f e \models_{f,n} [\![(\Delta, \mu, \sigma)]\!]$, if:*

$$\begin{aligned} &valid_f\ \theta\ \phi \rightarrow \\ &\quad P_{stat} \wedge (\forall E\ h\ k\ h'\ v\ \eta\ \delta\ m\ s\ \overline{s_i}^n \,.\, P_{\Downarrow}(f, n) \wedge P_{dyn} \wedge P_{size} \wedge P_\eta \rightarrow P_\Delta \wedge P_\mu \wedge P_\sigma) \end{aligned}$$

Definition 13. *A global bound environment Σ is* valid *up to depth n for f, denoted $\models_{f,n} \Sigma$, if it belongs to the following inductively defined set:*

1. *An environment in which f is not defined is valid at any depth for f if it is valid in general, i.e. if $\models \Sigma$ and $f \notin dom\ \Sigma$, then $\models_{f,n} \Sigma$.*
2. *A valid environment can be extended with any bound for f at depth 0, i.e. for all (Δ, μ, σ), if $\models \Sigma$ then $\models_{f,0} \Sigma \uplus \{f \mapsto (\Delta, \mu, \sigma)\}$.*
3. *If $\models \Sigma$, function f is defined as $f\ \overline{x_i}^l\ @\ \overline{r_j}^m = e_f$, and θ, ϕ are valid local environments for f, and $\theta, \phi, (l + m) \rhd_f e_f \models_{f,n} [\![(\Delta, \mu, \sigma)]\!]$ holds, then $\models_{f,n+1} \Sigma \uplus \{f \mapsto (\lfloor \Delta \rfloor, \mu, \sigma)\}$ also holds.*
4. *If $\models_{f,n} \Sigma$, function $g \neq f$ is defined as $g\ \overline{y_i}^l\ @\ \overline{r_j}^m = e_g$, and θ, ϕ are valid local environments for g, and $\theta, \phi, (l + m) \rhd_g e_g \models [\![(\Delta, \mu, \sigma)]\!]$ holds, then $\models_{f,n} \Sigma \uplus \{g \mapsto (\lfloor \Delta \rfloor, \mu, \sigma)\}$ also holds.*

Definition 14. *Given a context function f, we say that an expression e* conditionally satisfies *a bound (Δ, μ, σ), up tp depth n for f, with respect to a bound environment Σ in the context of θ, ϕ, denoted $\theta, \phi, td \rhd_f e, \Sigma \models_{f,n} [\![(\Delta, \mu, \sigma)]\!]$, if*

$$\models_{f,n} \Sigma \to \theta, \phi, td \rhd_f e \models_{f,n} [\![(\Delta, \mu, \sigma)]\!]$$

The following lemmas relate satisfaction and validity at depth n with satistacfion and validity in general.

Lemma 2. $\forall n \,.\, \theta, \phi, td \rhd_f e \models_{f,n} [\![(\Delta, \mu, \sigma)]\!] \quad \textit{iff} \quad \theta, \phi, td \rhd_f e \models [\![(\Delta, \mu, \sigma)]\!]$.

Proof. By equational reasoning.

$\forall n \,.\, \theta, \phi, td \rhd_f e \models_{f,n} [\![(\Delta, \mu, \sigma)]\!]$
$\equiv$ {By Def. 12}
$\forall n \,.\, (valid_f\ \theta\ \phi \to P_{stat} \wedge (\forall E\ h\ k\ h'\ v\ \eta\ \delta\ m\ s\ \overline{s_i} \,.\, P_{\Downarrow}(f, n) \wedge P_{dyn} \wedge P_{size} \wedge P_{\eta}$
$\to P_{\Delta} \wedge P_{\mu} \wedge P_{\sigma}))$
$\equiv$ {By first-order logic}
$valid_f\ \theta\ \phi \to P_{stat} \wedge (\forall E\ h\ k\ h'\ v\ \eta\ \delta\ m\ s\ \overline{s_i} \,.\, (\exists n \,.\, P_{\Downarrow}(f, n)) \wedge P_{dyn} \wedge P_{size} \wedge P_{\eta}$
$\equiv$ {By Lemma 1}
$valid_f\ \theta\ \phi \to P_{stat} \wedge (\forall E\ h\ k\ h'\ v\ \eta\ \delta\ m\ s\ \overline{s_i} \,.\, P_{\Downarrow} \wedge P_{dyn} \wedge P_{size} \wedge P_{\eta} \to P_{\Delta} \wedge P_{\mu} \wedge P_{\sigma})$
$\equiv$ {By Def. 8}
$\theta, \phi, td \rhd_f e \models [\![(\Delta, \mu, \sigma)]\!]$ □

Lemma 3. $\forall n \,.\, \models_{f,n} \Sigma \quad \textit{iff} \quad \models \Sigma$.

Proof. We separatedly prove each direction:

($\Leftarrow$) By induction on the $\models$ relation. If $\Sigma = \emptyset$ then we must prove $\forall n \,.\, \models_{f,n} \emptyset$, which is trivial by Def. 13-(1). If $\Sigma = \Sigma' \uplus \{f' \mapsto (\Delta, \mu, \sigma)\}$ then must prove:

$$\models \Sigma' \uplus \{f' \mapsto (\Delta, \mu, \sigma)\} \Rightarrow \forall n \,.\, \models_{f,n} \Sigma' \uplus \{f' \mapsto (\Delta, \mu, \sigma)\}$$

By applying Def. 9-(2) of $\models$, this es equivalent to proving:

$$\models \Sigma' \wedge \theta, \phi, (l + m) \rhd_{f'} e_{f'} \models [\![(\Delta', \mu, \sigma)]\!] \Rightarrow \forall n \,.\, \models_{f,n} \Sigma' \uplus \{f' \mapsto (\Delta, \mu, \sigma)\}$$

with $\Delta = \lfloor \Delta' \rfloor$ and valid θ, ϕ. By induction hypothesis, we can also assume $\forall n \,.\, \models_{f,n} \Sigma'$. Now, we proceed by cases on f':

$f' = f$ We proceed by induction on n:

- If $n = 0$ we must prove $\models_{f,0} \Sigma' \uplus \{f \mapsto (\Delta, \mu, \sigma)\}$, which is trivial by Def. 13-(2).
- For $n+1$ we must prove $\models_{f,n+1} \Sigma' \uplus \{f \mapsto (\Delta, \mu, \sigma)\}$. By Def. 13-(3), this is equivalent to proving $\models \Sigma' \wedge \theta, \phi, (l + m) \rhd_f e_f \models_{f,n} [\![(\Delta'', \mu, \sigma)]\!]$, for a Δ'' such that $\Delta = \lfloor \Delta'' \rfloor$, and valid θ, ϕ. But we have as hypothesis $\models \Sigma' \wedge \theta, \phi, (l + m) \rhd_f e_f \models [\![(\Delta', \mu, \sigma)]\!]$ with $\Delta = \lfloor \Delta' \rfloor$, which is stronger by Lemma 2.

$f' \neq f$ We must prove $\forall n.\models_{f,n} \Sigma' \uplus \{f' \mapsto (\Delta,\mu,\sigma)\}$. By Def. 13-(4), this is equivalent to proving $(\forall n\,.\models_{f,n})\Sigma' \wedge \theta,\phi,(l+m) \rhd_{f'} e_{f'} \models [\![(\Delta',\mu,\sigma)]\!]$ with $\Delta = \lfloor\Delta'\rfloor$. The first conjunct holds by induction hypothesis, and the second one by hypothesis.

($\Rightarrow$) We distinguish here two cases:

$f \notin dom\ \Sigma$ By doing induction on $\models_{f,n}$ we get four cases corresponding to those of Def. 13, all of them trivial.

$f \in dom\ \Sigma$ We must prove $f \in dom\ \Sigma \wedge (\forall n.\models_{f,n} \Sigma) \Rightarrow \models \Sigma$. We will prove the slightly stronger property $f \in dom\ \Sigma \wedge (\forall n > 0\,.\models_{f,n} \Sigma) \Rightarrow \models \Sigma$. The proof is split into two pieces:

1. $f \in dom\ \Sigma \wedge \forall n > 0\,.\models_{f,n} \Sigma \Rightarrow \forall n\,.\theta,\phi,(l+m) \rhd_f e_f \models_{f,n} [\![(\Delta',\mu,\sigma)]\!]$
2. $f \in dom\ \Sigma \wedge \forall n > 0\,.\models_{f,n} \Sigma \wedge \theta,\phi,(l+m) \rhd_f e_f \models [\![(\Delta',\mu,\sigma)]\!] \Rightarrow \models \Sigma$

By Lemma 2, the conclusion of the first one implies the third premise of the second one. Both proofs can easily be done by induction on the $\models_{f,n}$ relation.

Lemma 4.

If $\forall n\,.\theta,\phi,td \rhd_f e,\Sigma \models_{f,n} [\![(\Delta,\mu,\sigma)]\!]$ *then* $\theta,\phi,td \rhd_f e,\Sigma \models [\![(\Delta,\mu,\sigma)]\!]$

Proof. By equational reasoning.

$$
\begin{array}{ll}
 & \forall n\,.\theta,\phi,td \rhd_f e,\Sigma \models_{f,n} [\![(\Delta,\mu,\sigma)]\!] \\
\equiv & \quad \{\text{By Def. 14}\} \\
 & \forall n\,.(\models_{f,n} \Sigma \rightarrow \theta,\phi,td \rhd_f e \models_{f,n} [\![(\Delta,\mu,\sigma)]\!]) \\
\Rightarrow & \quad \{\text{By first-order logic}\} \\
 & (\forall n\,.\ \models_{f,n} \Sigma) \rightarrow (\forall n\,.\theta,\phi,td \rhd_f e \models_{f,n} [\![(\Delta,\mu,\sigma)]\!]) \\
\equiv & \quad \{\text{By lemmas 2 and 3}\} \\
 & \models \Sigma \rightarrow \theta,\phi,td \rhd_f e \models [\![(\Delta,\mu,\sigma)]\!] \\
\equiv & \quad \{\text{By Def. 10}\} \\
 & \theta,\phi,td \rhd_f e,\Sigma \models [\![(\Delta,\mu,\sigma)]\!] \qquad \square
\end{array}
$$

Having proved Lemma 4, the soundness of $\theta,\phi,td \rhd_f e,\Sigma \vdash (\Delta,\mu,\sigma)$, stated as Theorem 1, can be completed as follows:

Lemma 5 (Soundness).

If $\theta,\phi,td \rhd_f e,\Sigma \vdash (\Delta,\mu,\sigma)$ *then* $\forall n\,.\theta,\phi,td \rhd_f e,\Sigma \models_{f,n} [\![(\Delta,\mu,\sigma)]\!]$

Proof. By induction on the $\vdash$ derivation, and by cases on the last rule applied.

Lit See Lemma 6 in Section 5.2
Var See Lemma 7 in Section 5.2
Primop See Lemma 8 in Section 5.2
Cons See Lemma 9 in Section 5.2
Let See Lemma 10 in Section 5.2
Case See Lemma 11 in Section 5.2

App See Lemma 12 in Section 5.2
Rec Let us assume that the last rule applied was *Rec*. On the one hand, we have:

$$
\begin{array}{ll}
& \theta, \phi, l + m \rhd_f e_f, \Sigma \vdash (\Delta', \mu', \sigma') \\
\equiv & \quad \{\text{By the } \mathit{REC} \text{ rule}\} \\
& f\ \overline{x_i^{\,l}}@\ \overline{r_j^{\,m}} = e_f \wedge \theta, \phi, l + m \rhd_f e_f, \Sigma \uplus \{f \mapsto (\Delta, \mu, \sigma)\} \vdash (\Delta', \mu', \sigma') \wedge \\
& (\lfloor \Delta' \rfloor, \mu', \sigma') \sqsubseteq (\Delta, \mu, \sigma) \\
\Rightarrow & \quad \{\text{By induction hypothesis on } \vdash\} \\
& \forall n\,.\,\theta, \phi, l + m \rhd_f e_f, \Sigma \uplus \{f \mapsto (\Delta, \mu, \sigma)\} \models_{f,n} [\![(\Delta', \mu', \sigma')]\!] \\
\equiv & \quad \{\text{By Def. 14}\} \\
& \forall n\,.\,(\models_{f,n} \Sigma \uplus \{f \mapsto (\Delta, \mu, \sigma)\} \rightarrow \theta, \phi, l + m \rhd_f e_f \models_{f,n} [\![(\Delta', \mu', \sigma')]\!]) \qquad (1)
\end{array}
$$

On the other hand, we must prove:

$$\forall n\,.\,(\models_{f,n} \Sigma \rightarrow \theta, \phi, l + m \rhd_f e_f \models_{f,n} [\![(\Delta', \mu', \sigma')]\!])$$

We proceed by induction on n:

$n = 0$ By $f \notin \mathit{dom}\ \Sigma$ and Def. 13-(1), assuming $\models_{f,0} \Sigma$ is equivalent to assuming $\models \Sigma$, and this is in turn equivalent by Def. 13-(2) to $\models_{f,0} \Sigma \uplus \{f \mapsto (\Delta, \mu, \sigma)\}$. By (1) we get $\theta, \phi, l + m \rhd_f e_f \models_{f,0} [\![(\Delta', \mu', \sigma')]\!])$ and we are done.

$n > 0$ By Def. 13-(1), assuming $\models_{f,n+1} \Sigma$ is equivalent to assuming $\models \Sigma$, and this is in turn equivalent to $\models_{f,n} \Sigma$. By induction hypothesis on n we get $\theta, \phi, l + m \rhd_f e_f \models_{f,n} [\![(\Delta', \mu', \sigma')]\!])$. By $(\lfloor \Delta' \rfloor, \mu', \sigma') \sqsubseteq (\Delta, \mu, \sigma)$ we get $\theta, \phi, l + m \rhd_f e_f \models_{f,n} [\![(\lceil \Delta \rceil, \mu, \sigma)]\!])$, where we define $\lceil \Delta \rceil$ as Δ completed with $\{\rho^f_{self} \mapsto \Delta'\ \rho^f_{self}\}$. Obviously, $\Delta' \sqsubseteq \lceil \Delta \rceil$ and $\lfloor \lceil \Delta \rceil \rfloor = \Delta$. From this and from $\models \Sigma$, by Def. 13-(3) we get $\models_{f,n+1} \Sigma \uplus \{f \mapsto (\Delta, \mu, \sigma)\}$, and from (1) we get $\theta, \phi, l + m \rhd_f e_f \models_{f,n+1} [\![(\Delta', \mu', \sigma')]\!]$ as desired.

Notice that both $\models \Sigma \uplus \{f \mapsto (\Delta, \mu, \sigma)\}$ and $\models \Sigma \uplus \{f \mapsto (\lfloor \Delta' \rfloor, \mu', \sigma')\}$ can be easily obtained in the above proof. So, both are correct signatures for f. □

Then, the lemmas 4 and 5 complete the proof of the Soundness Theorem 1.

5.2 Proofs of the individual lemmas

Lit

Lemma 6. *The formula to be proved is:*

$$\mathit{If}\ \ \theta, \phi, td \rhd_f c, \Sigma \vdash ([\,]_f, 0, 1)\ \ \mathit{then}\ \ \forall n\,.\,\theta, \phi, td \rhd_f c, \Sigma \models_{f,n} [\![([\,]_f, 0, 1)]\!]$$

Proof. The *Lit* proof-rule has no premises, so we must prove Lemma's conclusion from scratch. Let us assume an arbitrary but fixed n. We must prove:

$$\mathit{valid}_f\ \theta\ \phi \rightarrow P_{stat} \wedge (\forall E\ h\ k\ h'\ v\ \eta\ \delta\ m\ s\ \overline{s_i}^n . P_{\Downarrow}(f, n) \wedge P_{dyn} \wedge P_{size} \wedge P_\eta \rightarrow P_\Delta \wedge P_\mu \wedge P_\sigma)$$

The premise $\mathit{valid}_f\ \theta\ \phi$ is not needed in this case. P_{stat} holds by definition of $[\,]_f$. Let us assume $P_{\Downarrow}(f, n)$ for appropriate $E\ h\ k\ \eta\ \overline{s_i}^n$ satisfying $P_{dyn} \wedge P_{size} \wedge P_\eta$. From the *Lit* semantic rule we get $(\delta, m, s) = ([\,]_k, 0, 1)$. Trivially it holds that $\forall j \in \{0 \ldots k\}\,.\,\sum_{\eta\ \rho = j}\ [\,]_f\ \rho\ \overline{s_i}^n = 0 \geq 0 = \delta\ j$, and $\mu\ \overline{s_i}^n = 0 \geq 0 = m$, and $\sigma\ \overline{s_i}^n = 1 \geq 1 = s$. So, we get $P_\Delta \wedge P_\mu \wedge P_\sigma$ as desired.

Var

Lemma 7. *The formula to be proved is:*

$$\textit{If}\ \ \theta, \phi, td \rhd_f x, \Sigma \vdash ([\,]_f, 0, 1)\ \ \textit{then}\ \ \forall n\,.\,\theta, \phi, td \rhd_f x, \Sigma \models_{f,n} [\![([\,]_f, 0, 1)]\!]$$

Proof. The proof is completely identical to the case *Lit*. □

Primop

Lemma 8. *The formula to be proved is:*

$$\textit{If}\ \theta, \phi, td \rhd_f a_1 \oplus a_2, \Sigma \vdash ([\,]_f, 0, 2)\ \ \textit{then}\ \ \forall n.\theta, \phi, td \rhd_f a_1 \oplus a_2, \Sigma \models_{f,n} [\![([\,]_f, 0, 2)]\!]$$

Proof. The proof is almost identical to the cases *Lit* and *Var*, the only diference being that a bound $\lambda\overline{x}.2$ is obtained for σ, and a stack consumption 2 for s. □

Cons

Lemma 9. *The formula to be proved is:*

$$\begin{array}{l}\textit{If}\ \ \theta, \phi, td \rhd_f C\ \overline{a_i}^n\ @\ r, \Sigma \vdash ([\theta\ r \mapsto 1], 1, 1)\\ \textit{then}\ \ \forall n\,.\,\theta, \phi, td \rhd_f C\ \overline{a_i}^n\ @\ r, \Sigma \models_{f,n} [\![([\theta\ r \mapsto 1]_f, 1, 1)]\!]\end{array}$$

Proof. Let us assume an arbitrary but fixed n. As the *Cons* proof-rule has no premises, we must prove from scratch:

$$\mathit{valid}_f\ \theta\ \phi \rightarrow P_{stat} \wedge (\forall E\ h\ k\ h'\ v\ \eta\ \delta\ m\ s\ \overline{s_i}.P_{\Downarrow}(f,n) \wedge P_{dyn} \wedge P_{size} \wedge P_\eta \rightarrow P_\Delta \wedge P_\mu \wedge P_\sigma)$$

Let us assume $\mathit{valid}_f\ \theta\ \phi$. Property P_{stat} holds by definition of $[\theta\ r \mapsto 1]_f$. Let us assume an evaluation $P_{\Downarrow}(f,n)$ of the expression with appropriate $E\ h\ k\ h'\ v\ \eta\ \overline{s_i}$ such that $P_{dyn} \wedge P_{size} \wedge P_\eta$ hold. From the semantic rule *Cons* we get $(\delta, m, s) = ([j \mapsto 1]_k, 1, 1)$. From $\mathit{consistent}(\theta, \eta, E, h))$, implied by $\mathit{valid}_f\ \theta\ \phi$, we get $j = E\ r = \eta\ (\theta\ r)$. Then, $P_\Delta \stackrel{\text{def}}{=} \forall j \in \{0 \ldots k\}\,.\,\sum_{\eta\ \rho=j}\ \Delta\ \rho\ \overline{s_i}^n \geq \delta\ j$ reduces to proving $1 \geq 1$. Properties P_μ and P_σ are also trivial. □

Let

Lemma 10. *The formula to be proved is:*

$$\textit{If}\ \ \theta, \phi, td \rhd_f \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, \Sigma \vdash (\Delta_1 + \Delta_2, \sqcup\{\mu_1, |\Delta_1| + \mu_2\}, \sqcup\{2 + \sigma_1, 1 + \sigma_2\})\ \ \textit{then}$$
$$\forall n\,.\,\theta, \phi, td \rhd_f \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, \Sigma \models_{f,n} [\![(\Delta_1 + \Delta_2, \sqcup\{\mu_1, |\Delta_1| + \mu_2\}, \sqcup\{2 + \sigma_1, 1 + \sigma_2\})]\!]$$

Proof. By the *Let* proof rule of the derivation relation $\vdash$ we have:

$$\theta, \phi, 0 \rhd_f e_1, \Sigma \vdash (\Delta_1, \mu_1, \sigma_1)\ \ \text{and}\ \ \theta, \phi, td + 1 \rhd_f e_2, \Sigma \vdash (\Delta_2, \mu_2, \sigma_2)$$

By induction hypothesis on the statement of Lemma 5 we get:

$$\begin{array}{l}\forall n\,.\,\theta, \phi, 0 \rhd_f e_1, \Sigma \models_{f,n} [\![(\Delta_1, \mu_1, \sigma_1)]\!]\ \ \text{and}\\ \forall n\,.\,\theta, \phi, td + 1 \rhd_f e_2, \Sigma \models_{f,n} [\![(\Delta_2, \mu_2, \sigma_2)]\!]\end{array}$$

Let us choose an arbitrary but fixed n for the whole proof. The steps are the following:

1. Assuming $valid_f\ \theta\ \phi$ for **let**, we have $valid_f\ \theta\ \phi$ for both e_1 and e_2. Then, we get P_{stat} and the rest of properties of $\_ \rhd \_\ \_ \models \_\ [\![(\_)]\!]$ for e_1 and e_2.
2. P_{stat} for e_1 and e_2 easily lead to P_{stat} for **let**.
3. Assuming $P_{\Downarrow}(f, n)$ for **let**, and using the semantic rule *Let*, we get $P_{\Downarrow}(f, n)$ for both e_1 and e_2 and appropriate heaps and runtime environments.
4. Likewise, assuming $P_{dyn} \wedge P_{size} \wedge P_{\eta}$ for **let**, it is straightforward to show $P_{dyn} \wedge P_{size} \wedge P_{\eta}$ for e_1 and e_2 on their respective heaps and runtime environments.
5. Then, we get $P_{\Delta} \wedge P_{\mu} \wedge P_{\sigma}$ for the bound $(\Delta_1, \mu_1, \sigma_1)$ with respect to the runtime consumption (δ_1, m_1, s_1) of e_1's evaluation, and for the bound $(\Delta_2, \mu_2, \sigma_2)$ with respect to e_2's consumption (δ_2, m_2, s_2).
6. Then, it only remains to be shown that $(\Delta_1 + \Delta_2, \sqcup\{\mu_1, |\Delta_1| + \mu_2\}, \sqcup\{2 + \sigma_1, 1 + \sigma_2\})$ is a bound for $(\delta_1 + \delta_2, \sqcup\{m_1, |\delta_1| + m_2\}, \sqcup\{2 + s_1, 1 + s_2\})$, which is trivial.

□

Case

Lemma 11. *The formula to be proved is:*

If $\theta, \phi, td \rhd_f$ ***case*** x ***of*** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{m}, \Sigma \vdash (\bigsqcup_{i=1}^{n} \Delta_i, \bigsqcup_{i=1}^{n} \mu_i, \bigsqcup_{i=1}^{n} (\sigma_i + n_i))$ *then*
$\forall n\ .\ \theta, \phi, td \rhd_f$ ***case*** x ***of*** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{m}, \Sigma \models_{f,n} [\![(\bigsqcup_{i=1}^{n} \Delta_i, \bigsqcup_{i=1}^{n} \mu_i, \bigsqcup_{i=1}^{n} (\sigma_i + n_i))]\!]$

Proof. By the *Case* proof rule of the derivation relation $\vdash$ we have:

$$(\forall i)\ \theta, \phi, td + n_i \rhd_f e_i, \Sigma \vdash (\Delta_i, \mu_i, \sigma_i)$$

By induction hypothesis on the statement of Lemma 5 we get:

$$(\forall i\ n)\ \theta, \phi, td + n_i \rhd_f e_i, \Sigma \models_{f,n} [\![(\Delta_i, \mu_i, \sigma_i)]\!]$$

Let us choose an arbitrary but fixed n for the whole proof. The steps are the following:

1. Assuming $valid_f\ \theta\ \phi$ for **case**, we have $valid_f\ \theta\ \phi$ for all the e_i. Then, we get P_{stat} and the rest of properties of $\_ \rhd \_\ \_ \models \_\ [\![(\_)]\!]$ for all the e_i.
2. P_{stat} for all the e_i lead to P_{stat} for **case**.
3. Assuming $P_{\Downarrow}(f, n)$ for **case**, and using the semantic rule *Case*, we get $P_{\Downarrow}(f, n)$ for one $e_j,\ j \in \{1..m\}$ for appropriate heap and runtime environment.
4. Likewise, assuming $P_{dyn} \wedge P_{size} \wedge P_{\eta}$ for **case**, it is straightforward to show $P_{dyn} \wedge P_{size} \wedge P_{\eta}$ for e_j on its respective heap and runtime environment.
5. Then, we get $P_{\Delta} \wedge P_{\mu} \wedge P_{\sigma}$ for the bound $(\Delta_j, \mu_j, \sigma_j)$ with respect to the runtime consumption (δ_j, m_j, s_j) of e_j's evaluation.
6. By knowing that, we must show that $(\bigsqcup_{i=1}^{n} \Delta_i, \bigsqcup_{i=1}^{n} \mu_i, \bigsqcup_{i=1}^{n} (\sigma_i + n_i))$ is a correct bound for $(\delta_j, m_j, s_j + n_j)$, which is straightforward.

□

App

Lemma 12. *The formula to be proved is:*

$$\begin{array}{l} \textit{If}\ \ \theta, \phi, td \rhd_f g\ \overline{a_i}^l\ @\ \overline{r_j}^q, \Sigma \vdash (\Delta, \mu, \sqcup\{l+q, \sigma - \ td + l + q\})\ \ \textit{then} \\ \forall n\,.\,\theta, \phi, td \rhd_f g\ \overline{a_i}^l\ @\ \overline{r_j}^q, \Sigma \models_{f,n} [\![(\Delta, \mu, \sqcup\{l+q, \sigma - \ td + l + q\})]\!] \end{array}$$

Proof. In the first place let us assume that $g \neq f$, and let us choose an arbitrary fixed n for the whole proof. By the *App* proof rule, we know:

$$\begin{array}{ll} \Sigma\ g = (\Delta_g, \mu_g, \sigma_g) & G\ \overline{x}^n \equiv \forall i \in \{1..l\}.\phi\ a_i\ \overline{x}^n \neq -\infty \\ argP(\psi, \overline{\rho'_j}^q, \theta, \overline{r_j}^q) & \mu = \lambda \overline{x}^n.[G\ \overline{x}^n \rightarrow \mu_g\ (\overline{\phi\ a_i\ \overline{x}^n}^l)] \\ \sigma = \lambda \overline{x}^n.[G\ \overline{x}^n \rightarrow \sigma_g\ (\overline{\phi\ a_i\ \overline{x}^n}^l)] & \Delta = instance_f(\Delta_g, \psi, \overline{a_i}^l) \end{array}$$

From Def. 14 we can assume $\models_{f,n} \Sigma$. We can also assume $valid_f\ \theta\ \phi$ as it is a premise of lemma's consequent. From Def. 13, and by doing induction on the $\models_{f,n}$ relation, we get that $\theta_g, \phi_g, (l+q) \rhd_g e_g \models_{f,n} [\![(\Delta'_g, \mu_g, \sigma_g)]\!]$ holds for some valid θ_g, ϕ_g, and a Δ'_g such that $\Delta_g = \lfloor \Delta'_g \rfloor$. Then, we get for e_g and the bound $(\Delta'_g, \mu_g, \sigma_g)$:

$$P_{stat} \wedge (\forall E\ h\ k\ h'\ v\ \eta\ \delta\ m\ s\ \overline{s_i}^n\,.\,P_{\Downarrow}(f,n) \wedge P_{dyn} \wedge P_{size} \wedge P_\eta \rightarrow P_\Delta \wedge P_\mu \wedge P_\sigma)$$

We must prove for $g\ \overline{a_i}^l\ @\ \overline{r_j}^q$ and the bound $(\Delta, \mu, \sqcup\{l+q, \sigma - \ td + l + q\})$ exactly the same properties. The steps are the following:

1. $P_{stat} \stackrel{\text{def}}{=} dom\ \Delta = R_f \cup \{\rho^f_{self}\}$ is just a consequence of the definition of the $instance_f$ function.
2. Let us assume $P_{\Downarrow}(f,n)$ for the application and some particular E, h, h', k, v, η, δ, m, s, and $\overline{s_i}^n$ such that $P_{dyn} \wedge P_{size} \wedge P_\eta$ hold.
3. By the *App* semantics' rule, and assuming that $g\ \overline{y_i}^l\ @\ \overline{r'_j}^q = e_g$ is the definition for g, we get:

   $$E_g \vdash h, k+1, l+q, e_g \Downarrow_{f,n} h', k+1, v, (\delta_g, m_g, s_g)$$

   where $E_g = [\overline{y_i \mapsto E\ a_i}^l, \overline{r'_j \mapsto E\ r_j}^q, self \mapsto k+1]$, $\delta = \delta_g|_k$, $m = m_g$, and $s = \sqcup\{l+q, s_g + l + q - td\}$. Then, we have $P_{\Downarrow}(f,n)$ for these values.
4. Now we choose $\eta_g = (\eta \cdot \psi) \uplus \{\rho^g_{self} \mapsto k{+}1\}$, and $\overline{s'_i}^l = \overline{E_g\ y_i}^l = \overline{E\ a_i}^l$. Having $argP(\psi, \overline{\rho'_j}^q, \theta, \overline{r_j}^q)$, it is easy to show that these values satisfy $P_{dyn} \wedge P_{size} \wedge P_\eta$ for e_g.
5. Then, we get $P_\Delta \wedge P_\mu \wedge P_\sigma$ for η_g, $k+1$, (δ_g, m_g, s_g), $\overline{s'_i}^l$, and the bound $(\Delta'_g, \mu_g, \sigma_g)$. It remains to be proved that $(\Delta, \mu, \sqcup\{l+q, \sigma + l + q - td\})$ is a bound for k, η, (δ, m, s), and $\overline{s_i}^n$.
6. Let us prove P_Δ. By $valid_f\ \theta\ \phi$ and by Def. 7, for each $i \in \{1 \ldots l\}$ we have:

   $$\phi\ a_i\ \overline{s_i}^n \geq size(h, E\ a_i) = size(h, E_g\ y_i) = s'_i \qquad (2)$$

   which is always positive, i.e. $\phi\ a_i\ \overline{s_i}^n \neq -\infty$. So, by the definition of $instance_f$, and for all $\rho \in R_f \cup \{\rho^f_{self}\}$, all $\rho' \in R_g$, and all $j \in \{0 \ldots k\}$ we get:

   $$\sum_{\eta\ \rho=j} \Delta\ \rho\ \overline{s_i}^n = \sum_{\eta\ \rho=j}\ \sum_{\psi\ \rho'=\rho} \Delta_g\ \rho'\ \overline{\phi\ a_i\ \overline{s_i}^n}^l$$

Because of the monotonicity of Δ_g we get:

$$\sum_{\eta\ \rho=j} \Delta\ \rho\ \overline{s_i}^n \geq \sum_{\eta\ \rho=j}\ \sum_{\psi\ \rho'=\rho} \Delta_g\ \rho'\ \overline{s'_i}^l = \sum_{(\eta\cdot\psi)\ \rho'=j} \Delta_g\ \rho'\ \overline{s'_i}^l \geq \delta\ j$$

because $\Delta_g = \lfloor \Delta'_g \rfloor$, and $\eta_g = \eta \cdot \psi$ for $\rho' \in R_g$, and Δ'_g is a bound for δ_g, and $\delta = \delta_g|_k$.

7. Let us prove P_μ. The steps are:

$$\begin{array}{lll} \mu\ \overline{s_i}^n = \mu_g\ \overline{\phi\ a_i\ \overline{s_i}^n}^l & & \{\text{because } G\ \overline{x_i}^n \text{ is true}\} \\ \geq \mu_g\ \overline{s'_i}^l & & \{\text{because of (2) and monotonicity of } \mu_g\} \\ \geq m_g & & \{\text{because } \mu_g \text{ is a bound for } m_g\} \\ = m & & \end{array}$$

8. Finally, we prove P_σ. By $G\ \overline{x_i}^n$ true, and σ_g being monotonic and a bound for s_g we get:

$$\begin{array}{l} \sigma\ \overline{s_i}^n = \sqcup\ \{l+q, \sigma_g\ \overline{(\phi\ a_i|\ \overline{s_j}^n)}^l + l + q - td\} \\ \geq \sqcup\ \{l+q, \sigma_g\ \overline{s'_j}^l + l + q - td\} \\ \geq \sqcup\ \{l+q, s_g + l + q - td\} \\ = s \end{array}$$

Let us consider now $g = f$. We must prove:

$$\forall n\,.\,P_{stat} \wedge (\forall E\ h\ k\ h'\ v\ \eta\ \delta\ m\ s\ \overline{s_i}^n\,.\,P_{\Downarrow}(f,n) \wedge P_{dyn} \wedge P_{size} \wedge P_\eta \rightarrow P_\Delta \wedge P_\mu \wedge P_\sigma)$$

for the application $f\ \overline{a_i}^l\ @\ \overline{r_j}^q$ and the bounds $(\Delta, \mu, \sqcup\{l+q, \sigma - \ td + l + q\})$. The reasoning for proving P_{stat} is the same as above. Now we distinguish two cases according to the value of n:

$n = 0$ In this case, by Def. 11 $P_{\Downarrow}(f,n)$ is false for the expression $f\ \overline{a_i}^l\ @\ \overline{r_j}^q$ (at least one call to f is being done). So the whole predicate is true.

$n > 0$ From Def. 14 we can assume $\models_{f,n} \Sigma$. We can also assume $valid_f\ \theta\ \phi$ as it is a premise of lemma's consequent. From Def. 13, and by doing induction on the $\models_{f,n}$ relation, we get that $\theta_f, \phi_f, (l+q) \rhd_f e_f \models_{f,n-1} [\![(\Delta'_f, \mu_f, \sigma_f)]\!]$ holds for any valid θ_f, ϕ_f, and a Δ'_f such that $\Delta_f = \lfloor \Delta'_f \rfloor$. Then, we have:

$$P_{stat} \wedge (\forall E\ h\ k\ h'\ v\ \eta\ \delta\ m\ s\ \overline{s_i}\,.\,P_{\Downarrow}(f,n-1) \wedge P_{dyn} \wedge P_{size} \wedge P_\eta \rightarrow P_\Delta \wedge P_\mu \wedge P_\sigma)$$

for e_f and the bounds $(\Delta'_f, \mu_f, \sigma_f)$. But, by the *App* rule of the relation $\Downarrow_{(f,n)}$, if $P_{\Downarrow}(f,n)$ holds for the expression $f\ \overline{a_i}^l\ @\ \overline{r_j}^q$, then $P_{\Downarrow}(f,n-1)$ holds for the body e_f, the same heap with one additional empty region $k+1$, and the appropriate runtime environment E_f. The remaining steps are almost identical to the steps (4) to (8) of the case $g \neq f$ above. □

6 Certification

The proof-rules presented in Sec. 4 are valid whatever are the monotonic functions considered for describing sizes and costs. However, for certification purposes we restrict ourselves to the smaller class of monotonic **Max-Poly** functions:

Definition 15. *The class* ***Max-Poly*** *over* $\overline{x}^n$ *is the smallest set of expressions containing constants in* $\mathbb{R}^+$*, variables* $y \in \overline{x}^n$*, and closed under the operations* $\{+, *, \sqcup\}$*. We will call a max-poly to any element of* ***Max-Poly****.*

We will call a max-poly function to a function of the form $\lambda\overline{x}^n.p$ in $(\mathbb{R}^+)^n \to \mathbb{R}^+$, where p is a max-poly over $\overline{x}^n$.

Notice that all the three operations are commutative and associative, and that $+$ and $*$ distribute over $\sqcup$ in $\mathbb{R}^+$. The latter makes that any max-poly can be normalized to a form $p_1 \sqcup \ldots \sqcup p_n$, where all the p_i are ordinary polynomials. This property extends also to max-poly functions.

In our case and disregrading $+\infty$ (which in fact means absence of a bound), the size and cost functions return a value in $\mathbb{R}^+ \cup \{-\infty\}$. As they are monotonic, in each dimension i they return $-\infty$ in some (possibly empty) interval $[0..k_i)$, and when $(\forall i \,.\, x_i \geq k_i)$ they return a value greater than or equal to 0. This property can be expressed by a boolean guard on the x_i. Inspired by this, we restrict our elementary functions to have the form $[G\rightarrow f]$, where G is a guard of the form $\bigwedge_{i=1}^{n}(p_i \geq k_i)$, $k_i \in \mathbb{R}^+$, and all the p_i and f are multivariate max-polys over the set $\overline{x}^n$ of variables. The meaning of this *atomic guarded function* (AGF in what follows) is:

$$[G\rightarrow f] \stackrel{\text{def}}{=} \lambda\overline{x}^n \,.\, \begin{cases} -\infty & \text{if } \neg G \\ f & \text{if } G \end{cases}$$

Operating with AGFs satisfies the following properties (a, b, c denote AGFs):

1. $[G_1\rightarrow f_1] + [G_2\rightarrow f_2] = [G_1 \wedge G_2\rightarrow f_1 + f_2]$
2. $[G_1\rightarrow f_1] * [G_2\rightarrow f_2] = [G_1 \wedge G_2\rightarrow f_1 * f_2]$
3. $[G_1\rightarrow [G_2\rightarrow f]] = [G_1 \wedge G_2\rightarrow f]$
4. $(a \sqcup b) + c = (a + c) \sqcup (b + c)$
5. $(a \sqcup b) * c = (a * c) \sqcup (b * c)$

As a consequence, any function obtained by combining AGFs with $\{+, *, \sqcup\}$ can be normalized to:

$$[G_1\rightarrow f_1] \sqcup \ldots \sqcup [G_l\rightarrow f_l]$$

We will call it a *normalized AGF set*. Now, comming back to the proof-rules of figures 3 and 4, if we introduce in the environment Σ of the *Rec* rule a triple (Δ, μ, σ) consisting of normalized AGF sets, and then derive a triple (Δ', μ', σ'), the latter can be expressed also as normalized AGF sets. This is because the operations involved in the remaining proof-rules are $\{+, *, \sqcup\}$, and the instantiations of the *App* rule. The latter consists of substituting max-polys for variables inside a max-poly. The result will also be a max-poly.

So, the check $(\lfloor\Delta'\rfloor, \mu', \sigma') \sqsubseteq (\Delta, \mu, \sigma)$ of the *Rec* rule reduces to checking inequalities of the form:

$$[G_1\rightarrow f_1] \sqcup \ldots \sqcup [G_l\rightarrow f_l] \sqsubseteq [G'_1\rightarrow f'_1] \sqcup \ldots \sqcup [G'_m\rightarrow f'_m]$$

Assuming that all the AGFs are functions over $\overline{x}^n$, this is in turn equivalent to:

$$\forall \overline{x}^n \,.\, \bigwedge_{i=1}^{l} \bigvee_{j=1}^{m} [G_i \rightarrow f_i] \sqsubseteq [G'_j \rightarrow f'_j]$$

Then, the elementary operation is comparing two AGFs. This can be expressed as follows:

$$[G \rightarrow f] \sqsubseteq [G' \rightarrow f'] = G \rightarrow (G' \wedge f \leq f')$$

The comparison $f \leq f'$ consists of comparing two max-polys of the form $p_1 \sqcup \ldots \sqcup p_r$ and $q_1 \sqcup \ldots \sqcup q_s$, which we can decide by applying again the same idea:

$$f \leq f' = \bigwedge_{i=1}^{r} \bigvee_{j=1}^{s} p_i \leq q_j$$

Summarizing, to decide $(\lfloor \Delta' \rfloor, \mu', \sigma') \sqsubseteq (\Delta, \mu, \sigma)$ we generate first-order formulas in Tarski's theory of real closed fields [21]. It is well known that this theory is decidable, although the existent algorithms are not efficient at all. For instance, Collins' quantifier elimination algorithm [9], which is recognized to be a great improvement over the original Tarski's procedure, has still a worst case complexity polynomial in the maximum degree of the involved polynomials and doubly exponential in the number of quantified variables. It is implemented in several symbolic algebra tools such as Mathematica. We have used the QEPCAD system built by Collins' group [7] which contains an improved version of original Collins' algorithm.

Fortunately, the number of quantified variables in our case is the number of arguments of the *Safe* function being certified, and this is usually very small, typically from one to three. So for practical purposes the QEPCAD system, or a similar tool, can be used as a certificate checker. The *Safe* compiler is used, not only to generate the initial triple (Δ, μ, σ) for every *Safe* function, but also to derive the triple (Δ', μ', σ'), to normalize both, and eventually to generate the proof obligations in the form of Tarski's formulas. For the moment, the compiler and the QEPCAD system have not been directly connected.

7 Case Study

In Figure 5 we show the *Core-Safe* versions of the algorithms `merge` and `msort`, in which regions are explicit. We will explain in detail how the proof-rules are applied to `merge` (simpler, but it produces an uninteresting linear Tarski problem), and then will show in less detail the process for `msort` (which produces a more interesting quadratic one). Let us assume that the candidate memory bound obtained by the *Safe* compiler for *merge* live heap is:

$$\begin{aligned} \Delta_{merge}\ \rho = {} & [x \geq 2 \wedge y \geq 1 \rightarrow x + y - 2] && \text{-- } A \\ \sqcup\ & [x \geq 1 \wedge y \geq 2 \rightarrow x + y - 2] && \text{-- } B \\ \sqcup\ & [x \geq 1 \wedge y \geq 1 \rightarrow 0] && \text{-- } C \end{aligned}$$

```
merge x y @ r = case x of
                  []    -> y
                  ex:x' -> case y of
                             []    -> x
                             ey:y' -> let c = ex <= ey in
                                      case c of
                                       True  -> let z1 = merge x' y @ r in
                                                ex:z1 @ r
                                       False -> let z2 = merge x y' @ r in
                                                ey:z2 @ r

msort x @ r = case x of
                []    -> x
                ex:x' -> case x' of
                           []  -> x
                           _:_ -> let (x1,x2) = unshuffle x @ self self in
                                  let z1      = msort x1 @ r            in
                                  let z2      = msort x2 @ r            in
                                  merge z1 z2 @ r
```

Fig. 5. functions *merge* and *msort* in *Core-Safe*

Remember that the constructor application proof-rule gets $[x \geq 1 \wedge y \geq 1 \rightarrow 1]$ charged to region ρ, the one for **let** asks for the addition of the involved Δ's, and the one for **case** asks for $\sqcup$ of the branches. All in all, we obtain as derived bound the following function:

$$\begin{aligned}
\Delta'_{merge}\ \rho = {} & [x \geq 1 \wedge y \geq 1 \rightarrow 0] \\
& \sqcup [x \geq 1 \wedge y \geq 1 \rightarrow 0] \\
& \sqcup (([x - 1 \geq 2 \wedge y \geq 1 \rightarrow x - 1 + y - 2] \sqcup [x - 1 \geq 1 \wedge y \geq 2 \rightarrow x - 1 + y - 2] \\
& \quad \sqcup [x - 1 \geq 1 \wedge y \geq 1 \rightarrow 0]) + [x \geq 1 \wedge y \geq 1 \rightarrow 1]) \\
& \sqcup (([x \geq 2 \wedge y - 1 \geq 1 \rightarrow x + y - 1 - 2] \sqcup [x \geq 1 \wedge y - 1 \geq 2 \rightarrow x + y - 1 - 2] \\
& \quad \sqcup [x \geq 1 \wedge y - 1 \geq 1 \rightarrow 0]) + [x \geq 1 \wedge y \geq 1 \rightarrow 1])
\end{aligned}$$

After normalization and simplification, we get:

$$\begin{aligned}
\Delta'_{merge}\ \rho = {} & [x \geq 1 \wedge y \geq 1 \rightarrow 0] && \text{-- } C' \\
& \sqcup [x \geq 3 \wedge y \geq 1 \rightarrow x + y - 2] \sqcup [x \geq 2 \wedge y \geq 1 \rightarrow 1] && \text{-- } A' \sqcup A'' \\
& \sqcup [x \geq 2 \wedge y \geq 2 \rightarrow x + y - 2] && \text{-- } D' \\
& \sqcup [x \geq 1 \wedge y \geq 3 \rightarrow x + y - 2] \sqcup [x \geq 1 \wedge y \geq 2 \rightarrow 1] && \text{-- } B' \sqcup B''
\end{aligned}$$

Obviously, for all x, y we get $C' \sqsubseteq C$, $A' \sqsubseteq A$, $B' \sqsubseteq B$, and both $D' \sqsubseteq A$ and $D' \sqsubseteq B$. It is also easy to convince ourselves that A'' is dominated by A and B'' is dominated by B. Then, the inequality $\lfloor \Delta'_{merge} \rfloor \sqsubseteq \Delta_{merge}$ holds.

The candidate `msort` live memory bound inferred by our compiler, assuming Δ_{merge} as above, and the following bound obtained for *unshuffle*:

$$\Delta_{unshuffle} = \begin{bmatrix} \rho_1 \mapsto [x \geq 2 \rightarrow x + 1] \sqcup [x \geq 1 \rightarrow 2] \\ \rho_2 \mapsto [x \geq 2 \rightarrow x] \sqcup [x \geq 1 \rightarrow 1] \end{bmatrix}$$

is

$$\Delta_{msort}\ \rho = [x \geq 2 \rightarrow \frac{4}{3}x^2 - 3x] \sqcup [x \geq 1 \rightarrow 0]$$

Introducing this candidate bound in the environment, applying the proof-rules, normalizing, and simplifying lead to:

$$\Delta'_{msort} = \begin{bmatrix} \rho \mapsto [x \geq 3 \rightarrow \frac{2}{3}x^2 - \frac{3}{2}x - \frac{17}{6}] \sqcup [x \geq 1 \rightarrow 0] \\ \rho_{self} \mapsto [x \geq 2 \rightarrow 2x + 1] \sqcup [x \geq 1 \rightarrow 3] \end{bmatrix}$$

Notice that the charges to the *self* region are not needed in the comparison $\lfloor \Delta'_{msort} \rfloor \sqsubseteq \Delta_{msort}$. The relevant inequality is then:

$$\forall x\,. \qquad \ldots \left(x \geq 3 \rightarrow x \geq 2 \wedge (\frac{2}{3}x^2 - \frac{3}{2}x - \frac{17}{6} \leq \frac{4}{3}x^2 - 3x) \right) \ldots$$

When this formula is given to QEPCAD, it answers *True* in about 100 msec. Then, $\lfloor \Delta'_{msort} \rfloor \sqsubseteq \Delta_{msort}$ holds.

8 Related Work and Conclusion

A seminal paper on static inference of memory bounds is [13]. A special type inference algorithm generates a set of linear constraints which, if satisfiable, they build a safe linear bound on the heap consumption. Afterwards, the authors extended this work to certificate generation [4], the certificate being an Isabelle/HOL proof-script which in essence was a proof of correctness of the type system, specialized for the types of the program being certified.

One of the authors extended in [12] the type system of [13] in order to infer polynomial bounds. Although not every polynomial could be inferred by this system, the work was a remarkable step forwards in the area. They do not pay attention to certificates in this paper but there is an occasional comment on that the same ideas of [4] could be applied here.

In [8] an abstract interpretation based algorithm for controlling that memory is not allocated inside loops in Java programs is verified by using the Coq proof-assistant [5]. Here there is no program-specific certificate, but a general proof of correctness of the analysis algorithm.

With respect to our proof-rules, they clearly have an abstract interpretation flavour. In fact variants of these rules have been used in [16] for inferring the candidate bounds. For recursive functions, we have adapted to our framework the technique first explained in [18] for verifying properties of recursive procedures. This technique has also been used in similar works (see e.g. [2]) where procedure global environments occur, and recursive procedures must be verified. The main idea is to explicitly introduce the depth of recursive call chains in the environment, and then doing some form of induction on this depth.

We have found inspiration on some work on quasi-interpretations for characterizing the complexity classes of rewriting systems [6], where **Max-Poly** play a role. The existence of a quasi-interpretation belonging to **Max-Poly** is used to decide that some systems are in the classes PTIME or PSPACE. They show that the problem is decidable by generating formulas in first-order Tarski's theory over the reals. The formulas are existentially quantified and they assert the existence of a quasi-interpretation, although no attempt to synthesize it is done.

Our work finds for the first time a way of separating the bound inference problem from the certification one. We have shown that certification need not be a kind of proof of correctness of the inference algorithm. It could be then applied to languages other than *Safe* (for instance, to the functional language used in [13] and [12]), where other algorithms and type systems are used to compute the candidate bounds.

Additionally, our language takes into account the memory deallocation due to the region mechanism. Most of other approaches infer and/or certify bounds to the *total* allocated memory, as opposed to the *live* and *peak* memory, respectively reached after and during program evaluation.

Acknowledgements: We are grateful to our colleague Maria Emilia Alonso for putting us on the tracks of the QEPCAD system.

References

1. C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In R. Cousot and M. Martel, editors, *SAS*, volume 6337 of *LNCS*, pages 117–133. Springer, 2010.
2. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resources. *Theoretical Computer Science*, 389:411–445, 2007.
3. A. M. Ben-Amram and M. Codish. A SAT-Based Approach to Size Change Termination with Global Ranking Functions. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 218–232. Springer, 2008.
4. L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In *LPAR'04, LNAI 3452*, pages 347–362, 2005.
5. Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
6. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations. Technical Report, Loria, http://www.loria.fr/~moyen, 2004.
7. C. W. Brown. QEPCAD: Quantifier Elimination by Partial Cylindrical Algebraic Decomposition. http://www.cs.usna.edu/qepcad/B/QEPCAD.html, 2004.
8. D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In *FM 2005, LNCS 3582*, pages 91–106. Springer, 2005.
9. G. E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In H. Barkhage, editor, *Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 134–183. Springer, 1975.
10. M. Colón and H. Sipma. Practical Methods for Proving Program Termination. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *LNCS*, pages 442–454. Springer, 2002.
11. 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.
12. J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. A Static Inference of Polynomial Bounds for Functional Programs. In *ESOP 2010, LNCS 6012*, pages 287–306. Springer, 2010.
13. 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.

14. S. Lucas and R. Peña. Rewriting Techniques for Analysing Termination and Complexity Bounds of SAFE Programs. In *Proc. Logic-Based Program Synthesis and Transformation, LOPSTR'08, Valencia, Spain*, pages 43–57, July 2008.
15. M. Montenegro, R. Peña, and C. Segura. A Simple Region Inference Algorithm for a First-order Functional Language. In S. Escobar, editor, *Work. on Functional and Logic Programming, WFLP 2009, Brasilia*, pages 145–161. LNCS 5979, 2009.
16. M. Montenegro, R. Peña, and C. Segura. A space consumption analysis by abstract interpretation. In *Selected papers of Foundational and Practical Aspects of Resource Analysis, FOPARA'09, Eindhoven*, pages 34–50. LNCS 6324, Nov. 2009.
17. G. C. Necula. Proof-Carrying Code. In *ACM SIGPLAN-SIGACT Principles of Programming Languages, POPL'97*, pages 106–119. ACM Press, 1997.
18. T. Nipkow. Hoare Logics for Recursive Procedures and Unbounded Nondeterminism. In *16th Int. Workshop on Computer Science Logic, CSL'02*, pages 103–119. LNCS 2471, Springer, 2002.
19. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.
20. A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.
21. A. Tarski. A Decision Method for Elementary Algebra and Geometry. University of California Press, Berkeley, 1948.