# A Space Consumption Analysis By Abstract Interpretation *

Manuel Montenegro, Ricardo Peña and Clara Segura
montenegro@fdi.ucm.es {ricardo,csegura}@sip.ucm.es

Universidad Complutense de Madrid, Spain

**Abstract.** *Safe* is a first-order functional language with an implicit region-based memory system and explicit destruction of heap cells. Its static analysis for inferring regions, and a type system guaranteeing the absence of dangling pointers have been presented elsewhere.
In this paper we present a new analysis aimed at inferring upper bounds for heap and stack consumption. It is based on abstract interpretation, being the abstract domain the set of all $n$-ary monotonic functions from real non-negative numbers to a real non-negative result. This domain turns out to be a complete lattice under the usual $\sqsubseteq$ relation on functions. Our interpretation is monotonic in this domain and the solution we seek is the least fixpoint of the interpretation.
We first explain the abstract domain and some correctness properties of the interpretation rules with respect to the language semantics, then present the inference algorithms for recursive functions, and finally illustrate the approach with the upper bounds obtained by our implementation for some case studies.

## 1 Introduction

The first-order functional language *Safe* has been developed in the last few years as a research platform for analysing and formally certifying two properties of programs related to memory management: absence of dangling pointers and having an upper bound to memory consumption. Two features make *Safe* different from conventional functional languages: (a) a region based memory management system which does not need a garbage collector; and (b) a programmer may ask for explicit destruction of memory cells, so that they could be reused by the program. These characteristics, together with the above certified properties, make *Safe* useful for programming small devices where memory requirements are rather strict and where garbage collectors are a burden in service availability.

The *Safe* compiler is equipped with a battery of static analyses which infer such properties [12, 13, 10]. These analyses are carried out on an intermediate language called *Core-Safe* explained below. We have developed a resource-aware operational semantics of *Core-Safe* [11] producing not only values but also exact figures on the heap and stack consumption of a particular running. The code generation phases have been certified in a proof assistant [5, 4], so that there is

* Work partially funded by the projects TIN2008-06622-C03-01/TIN (STAMP), S-0505/ TIC/ 0407 (PROMESAS) and the MEC FPU grant AP2006-02154.

a formal guarantee that the object code actually executed in the target machine (the JVM [9]) will exactly consume the figures predicted by the semantics.

Regions are dynamically allocated and deallocated. The compiler 'knows' which data lives in each region. Thanks to that, it can compute an upper bound to the space consumption of every region and so and upper bound to the total heap consumption. Adding to this a stack consumption analysis would result in having an upper bound to the total memory needs of a program.

In this work we present a static analysis aimed at inferring upper bounds for individual *Safe* functions, for expressions, and for the whole program. These have the form of $n$-ary mathematical functions relating the input argument sizes to the heap and stack consumption made by a *Safe* function, and include as particular cases multivariate polynomials of any degree. Given the complexity of the inference problem, even for a first-order language like *Safe*, we have identified three separate aspects which can be independently studied and solved: (1) Having an upper bound on the size of the call-tree deployed at runtime by each recursive *Safe* function; (2) Having upper bounds on the sizes of all the expressions of a recursive *Safe* function. These are defined as the number of cells needed by the normal form of the expression; and (3) Given the above, having an inference algorithm to get upper bounds for the stack and heap consumption of a recursive *Safe* function.

Several approaches to solve (1) and (2) have been proposed in the literature (see the Related Work section). We have obtained promising results for them by using rewriting systems termination proofs [10]. In case of success, these tools return multivariate polynomials of any degree as solutions. This work presents a possible solution to (3) by using abstract interpretation. It should be considered as a *proof-of-concept* paper: we investigate how good the upper bounds obtained by the approach are, provided we have the best possible solutions for problems (1) and (2). In the case studies presented below, we have introduced by hand the bounds to the call-tree and to the expression sizes.

The abstract domain is the set of all monotonic, non-negative, $n$-ary functions having real number arguments and real number result. This infinite domain is a complete lattice, and the interpretation is monotonic in the domain. So, fixpoints are the solutions we seek for the memory needs of a recursive *Safe* function. An interesting feature of our interpretation is that we usually start with an over-approximation of the fixpoint, but we can obtain tighter and tighter safe upper bounds just by iterating the interpretation any desired number of times.

The plan of the paper is as follows: Section 2 gives a brief description of our language; Section 3 introduces the abstract domain; Sections 4 and 5 give the abstract interpretation rules and some proof sketches about their correctness, while Section 6 is devoted to our inference algorithms for recursive functions; in Section 7 we apply them to some case studies, and finally in Section 8 we give some account on related and future work.

## 2 Safe in a Nutshell

*Safe* is polymorphic and has a syntax similar to that of (first-order) Haskell. In *Full-Safe* in which programs are written, regions are implicit. These are inferred when *Full-Safe* is desugared into *Core-Safe* [13]. The allocation and deallocation of regions is bound to function calls: a *working region* called *self* is allocated when entering the call and deallocated when exiting it. So, at any execution point only a small number of regions, kept in an invocation stack, are alive. The data structures built at *self* will die at function termination, as the following treesort algorithm shows:

```
treesort xs = inorder (mkTree xs)
```

First, the original list `xs` is used to build a search tree by applying function `mkTree` (not shown). The tree is traversed in inorder to produce the sorted list. The tree is not part of the result of the function, so it will be built in the working region and will die when the `treesort` function returns. The *Core-Safe* version of `treesort` showing the inferred type and regions is the following:

```
treesort :: [a] @ rho1 -> rho2 -> [a] @ rho2
treesort xs @ r = let t = mkTree xs @ self
                  in inorder t @ r
```

Variable `r` of type `rho2` is an additional argument in which `treesort` receives the region where the output list should be built. This is passed to the `inorder` function. However `self` is passed to `mkTree` to instruct it that the intermediate tree should be built in `treesort`'s *self* region.

Data structures can also be destroyed by using a destructive pattern matching, denoted by !, or by a **case!** expression, which deallocates the cell corresponding to the outermost constructor. Using recursion, the recursive portions of the whole data structure may be deallocated. As an example, we show a *Full-Safe* insertion function in an ordered list, which reuses the argument list's spine:

```
insertD x []! = x : []
insertD x (y:ys)! | x <= y = x : y : ys!
                  | x > y  = y : insertD x ys!
```

Expression $ys!$ means that the substructure pointed to by $ys$ in the heap is reused. The following is the (abbreviated) *Core-Safe* typed version:

```
insertD :: Int -> [Int]! @ rho -> rho -> [Int] @ rho
insertD x ys @ r = case! ys of
                    []   -> let zs = [] @ r in let us = (x:zs) @ r in us
                    y:yy -> let b = x <= y in case b of
                      True ->  let ys1 = (let yy1 = yy! in let as = (y:yy1) @ r in as) in
                               let rs1 = (x:ys1) @ r in rs1
                      False -> let ys2 = (let yy2 = yy! in insertD x yy2 @ r) in
                               let rs2 = (y:ys2) @ r in rs2
```

This function will run in constant heap space since, at each call, a cell is destroyed while a new one is allocated at region `r` by the (:) constructor. Only when the new element finds its place a new cell is allocated in the heap.

In Fig. 1 we show two *Core-Safe* big-step semantic rules in which a resource vector is obtained as a side effect of evaluating an expression. A judgement has the form $E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$ meaning that expression $e$ is evaluated in an environment $E$ using the $td$ topmost positions in the stack, and in a

$$\frac{\begin{array}{c}E \vdash h, k, 0, e_1 \Downarrow h', k, v_1, (\delta_1, m_1, s_1)\\ E \uplus [x_1 \mapsto v_1] \vdash h', k, td+1, e_2 \Downarrow h'', k, v, (\delta_2, m_2, s_2)\end{array}}{E \vdash h, k, td, \textbf{let } x_1 = e_1 \textbf{ in } e_2 \Downarrow h'', k, v, (\delta_1 + \delta_2, \max\{m_1, |\delta_1| + m_2\}, \max\{2 + s_1, 1 + s_2\})} \; [\mathit{Let_1}]$$

$$\frac{E\ x = p \quad C = C_r \quad E \uplus [\overline{x_{r_i} \mapsto v_i}^{n_r}] \vdash h, k, td + n_r, e_r \Downarrow h', k, v, (\delta, m, s)}{E \vdash h \uplus [p \mapsto (j, C\ \overline{v_i}^n)], k, td, \textbf{case!}\ x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^n \Downarrow h', k, v, (\delta + [j \mapsto -1], \max\{0, m-1\}, s + n_r)} \; [\mathit{Case!}]$$

**Fig. 1.** Two rules of the resource-aware operational semantics of *Safe*

heap $(h, k)$ with $0..k$ active regions. As a result, a heap $(h', k)$ and a value $v$ are obtained, and a resource vector $(\delta, m, s)$ is consumed. Notice that $k$ does not change because the number of active regions increases by one at each application and decreases by one at each function return, and all applications during $e$'s evaluation have been completed. A heap $h$ is a mapping between pointers 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 $\delta : \mathbb{N} \rightarrow \mathbb{Z}$ giving for each active region $i$ the signed difference between the cells in the final and initial heaps. A positive difference means that new cells have been created in this region. A negative one, means that some cells have been destroyed. By $dom(\delta)$ we denote the subset of $\mathbb{N}$ in which $\delta$ is defined. By $|\delta|$ we mean the sum $\sum_{n \in dom(\delta)} \delta(n)$ 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 $e$. When $e$ is the main expression, these figures give us the total memory needs of a particular run of the *Safe* program. For a full description of the semantics and the abstract machine see [11].

## 3 Function Signatures

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

$$\begin{array}{l} f :: t_1 \rightarrow \ldots t_n \rightarrow \rho_1 \rightarrow \ldots \rho_m \rightarrow t \\ f\ x_1 \cdots x_n\ @\ r_1\ \cdots r_m = e_f \end{array}$$

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. For example,

```
copy xs @ r = case xs of []   -> [] @ r
                         y:ys -> let zs = copy ys @ r in
                                 let rs = (y:zs) @ r in rs
```

charges as many cells to region $r$ as the input list size. We define the size of an algebraic type term to be the number of cells of its recursive spine and that of a boolean value to be zero. However, for a natural number we take its value because frequently space costs depend on the value of a numeric argument.

As a consequence, all the costs, sizes and needs of $f$ can be expressed as functions $\eta : (\mathbb{R}^+ \cup \{+\infty\})^n \rightarrow \mathbb{R} \cup \{+\infty, -\infty\}$ on $f$'s argument sizes. Infinite costs will be used to represent that we are not able to infer a bound (either because it does not exist or because the analysis is not powerful enough). Costs can be negative if the function destroys more cells than it builds. Currently we are restricting ourselves to functions where for each destructed cell at least a new cell is built in the same region. This covers many interesting functions where the aim of cell destruction is space reuse instead of pure destruction, e.g. function `insertD` shown in the previous section. This restriction means that the domain of the space cost functions is the following:

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

The domain $(\mathbb{F}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a complete lattice, where $\sqsubseteq$ is the usual order between functions, and the rest of components are standard. Notice that it is closed by the operations $\{+, \sqcup, *\}$. We abbreviate $\lambda \overline{x_i}^n.c$ by $c$, when $c \in \mathbb{R}^+$.

Function $f$ above may charge space costs to a maximum of $n+m+1$ regions: It may destroy cells in the regions where $x_1 \ldots x_n$ live; it may create/destroy cells in any output region $r_1 \ldots r_m$, and additionally in its *self* region. Each region $r$ has a region type $\rho$. We denote by $R^f_{in}$ the set of input region types, and by $R^f_{out}$ the set of output region types. For example, $R^{treesort}_{in} = \{\rho_1\}$ and $R^{treesort}_{out} = \{\rho_2\}$. Looked from outside, the charges to the *self* region are not visible, as this region disappears when the function returns.

Summarising, let $R_f = R^f_{in} \cup R^f_{out}$. Then $\mathbb{D} = \{\Delta : R_f \to \mathbb{F}\}$ is the complete lattice of functions that describe the space costs charged by $f$ to every visible region. In the following we will call abstract heaps to the functions $\Delta \in \mathbb{D}$.

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

The aim is that $\Delta_f$ describes (an upper bound to) the space costs charged by $f$ to every visible region, (i.e. the increment in live memory due to a call to $f$), and $\mu_f, \sigma_f$ respectively describe (an upper bound to) the heap and stack *needs* in order to execute $f$ without running out of space (i.e. the maximal increment in live memory during $f$'s evaluation). By $[\,]_f$ we denote the constant function $\lambda\rho.\lambda\overline{x_i}^n.0$, where we assume $\rho \in R_f$. By $|\Delta|$ we mean $\sum_{\rho \in dom(\Delta)} \Delta\ \rho$.

## 4 Abstract Interpretation

In Figure 2 we show the abstract interpretation rules for the most relevant *Core-Safe* expressions. There, an atom $a$ represents either a variable $x$ or a constant $c$, and $|e|$ denotes the function obtained by the size analysis for expression $e$. We can assume that the abstract syntax tree is decorated with such information.

When inferring an expression $e$, we assume it belongs to the body of a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$, that we will call the *context* function, and that only already inferred functions $g\ \overline{y_i}^l\ @\ \overline{r_j}^q = e_g$ are called. Let $\Sigma$ be a global environment giving, for each *Safe* function $g$ in scope, its signature $(\Delta_g, \mu_g, \sigma_g)$, let $\Gamma$ be a typing environment containing the types of all the variables appearing in $e_f$, and let $td$ be a natural number. The abstract interpretation $[\![e]\!]\ \Sigma\ \Gamma\ td$ gives a triple $(\Delta, \mu, \sigma)$ representing the space costs and needs of expression $e$. The statically determined value $td$ occurring as an argument of the interpretation and used in rule *App* is the size of the top part of the environment used

$$[\![a]\!]\ \Sigma\ \Gamma\ td = ([\,]_f, 0, 1) \quad [Atom]$$

$$[\![a_1 \oplus a_2]\!]\ \Sigma\ \Gamma\ td = ([\,]_f, 0, 2) \quad [Primop]$$

$$\frac{\begin{array}{c}\Sigma\ g = (\Delta_g, \mu_g, \sigma_g) \qquad \theta = unify\ \Gamma\ g\ \overline{a_i}^l\ \overline{r_j}^q \\ \mu = \lambda \overline{x}^n.\mu_g\ (\overline{|a_i|\ \overline{x}^n}^l) \quad \sigma = \lambda \overline{x}^n.\sigma_g\ (\overline{|a_i|\ \overline{x}^n}^l) \quad \Delta = \theta \downarrow_{\overline{|a_i|\ \overline{x}^n}^l} \Delta_g\end{array}}{[\![g\ \overline{a_i}^l\ @\ \overline{r_j}^q]\!]\ \Sigma\ \Gamma\ td = (\Delta, \mu, \sqcup\{l+q, \sigma - \ td + l + q\})} \ [App]$$

$$\frac{[\![e_1]\!]\ \Sigma\ \Gamma\ 0 = (\Delta_1, \mu_1, \sigma_1) \quad [\![e_2]\!]\ \Sigma\ \Gamma\ (td+1) = (\Delta_2, \mu_2, \sigma_2)}{[\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2]\!]\ \Sigma\ \Gamma\ td = (\Delta_1 + \Delta_2, \sqcup\{\mu_1, |\Delta_1| + \mu_2\}, \sqcup\{2+\sigma_1, 1+\sigma_2\})} \ [Let_1]$$

$$\frac{\Gamma\ r = \rho \quad [\![e_2]\!]\ \Sigma\ \Gamma\ (td+1) = (\Delta, \mu, \sigma)}{[\![\mathbf{let}\ x_1 = C\ \overline{a_i}^n\ @\ r\ \mathbf{in}\ e_2]\!]\ \Sigma\ \Gamma\ td = (\Delta + [\rho \mapsto 1], \mu + 1, \sigma + 1)} \ [Let_2]$$

$$\frac{(\forall i)\ [\![e_i]\!]\ \Sigma\ \Gamma\ (td + n_i) = (\Delta_i, \mu_i, \sigma_i)}{[\![\mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n]\!]\ \Sigma\ \Gamma\ td = (\bigsqcup_{i=1}^n \Delta_i, \bigsqcup_{i=1}^n \mu_i, \bigsqcup_{i=1}^n (\sigma_i + n_i))} \ [Case]$$

$$\frac{\Gamma\ x = T\overline{t_k}^l @\rho \quad (\forall i)\ [\![e_i]\!]\ \Sigma\ \Gamma\ (td + n_i) = (\Delta_i, \mu_i, \sigma_i)}{[\![\mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n]\!]\ \Sigma\ \Gamma\ td = ([\rho \mapsto -1] + \bigsqcup_{i=1}^n \Delta_i, \sqcup(0, \bigsqcup_{i=1}^n \mu_i - 1), \bigsqcup_{i=1}^n (\sigma_i + n_i))} \ [Case!]$$

**Fig. 2.** Space inference rules for expressions with non-recursive applications

when compiling the expression $g\ \overline{a_i}^l\ @\ \overline{r_j}^q$. This size is also an argument of the operational semantics. See [11] for more details.

Rules $[Atom]$ and $[Primop]$ exactly reflect the corresponding resource-aware semantic rules [11]. 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, $\overline{|a_i|\ \overline{x_j}^n}^l$ which have the $\overline{x}^n$ as free variables. Due to the application, some different region types of $g$ may instantiate to the same actual region type of $f$. That means that we must accumulate the memory consumed in some formal regions of $g$ in order to get the charge to an actual region of $f$. In Figure 2, $unify\ \Gamma\ g\ \overline{a_i}^l\ \overline{r_j}^q$ computes a substitution $\theta$ from $g$'s region types to $f$'s region types. If $\theta\ \rho_g = \rho_f$, this means that the generic $g$'s region type $\rho_g$ is instantiated to the $f$'s actual region type $\rho_f$. Formally, if $R_g = R^g_{in} \cup R^g_{out}$ then $\theta :: R_g \to R_f \cup \{\rho_{self}\}$ is total. The extension of region substitutions to types is straightforward.

**Definition 2.** *Given a type environment $\Gamma$, a function $g$ and the sequences $\overline{a_i}^l$ and $\overline{r_j}^q$, we say that $\theta = unify\ \Gamma\ g\ \overline{a_i}^l\ \overline{r_j}^q$ iff*

$$\Gamma\ g = \forall \overline{\alpha}.\overline{t_i}^l \to \overline{\rho_j}^q \to t \text{ and } \forall i \in \{1 \ldots l\}.\theta\ t_i = \Gamma\ a_i \text{ and } \forall j \in \{1 \ldots q\}.\theta\ \rho_j = \Gamma\ r_j$$

As an example, let us assume $g :: ([a]@\rho^g_1, [[b]@\rho^g_2]@\rho^g_1)@\rho^g_3 \to \rho^g_2 \to \rho^g_4 \to \rho^g_5 \to t$ and consider the application $g\ p\ @\ r_2\ r_1\ r_1$ where $p :: ([a]@\rho^f_1, [[b]@\rho^f_2]@\rho^f_1)@\rho^f_1$, $r_1 :: \rho^f_1$ and $r_2 :: \rho^f_2$. The resulting substitution would be:

$$\theta = [\rho^g_1 \mapsto \rho^f_1, \rho^g_2 \mapsto \rho^f_2, \rho^g_3 \mapsto \rho^f_1, \rho^g_4 \mapsto \rho^f_1, \rho^g_5 \mapsto \rho^f_1]$$

The function $\theta \downarrow_{\overline{\eta_i\ \overline{x}^n}^l} \Delta_g$ converts an abstract heap for $g$ into an abstract heap for $f$. It is defined as follows:

$$\theta \downarrow_{\overline{\eta_i\ \overline{x_j}^n}^l} \Delta_g = \lambda\rho\ .\ \lambda\overline{x_j}^n. \sum_{\substack{\rho' \in R_g \\ \theta\ \rho' = \rho}} \Delta_g\ \rho'\ \overline{\eta_i\ \overline{x_j}^n}^l \qquad (\rho \in R_f \cup \{\rho_{self}\}, \eta_i \in \mathbb{F})$$

In the example, we have:

$$\Delta\ \rho_2^f = \lambda \overline{x}^n.\Delta_g\ \rho_2^g\ \overline{(|a_i|\ \overline{x}^n)}^l$$
$$\Delta\ \rho_1^f = \lambda \overline{x}^n.\Delta_g\ \rho_1^g\ \overline{(|a_i|\ \ \overline{x}^n)}^l + \Delta_g\ \rho_3^g\ \overline{(|a_i|\ \ \overline{x}^n)}^l + \Delta_g\ \rho_4^g\ \overline{(|a_i|\ \overline{x}^n)}^l + \Delta_g\ \rho_5^g\ \overline{(|a_i|\ \ \overline{x}^n)}^l$$

Rules [*Let₁*] and [*Let₂*] reflect the corresponding resource-aware semantic rules in [11]. Rules [*Case*] and [*Case*!] use the least upper bound operators $\bigsqcup$ in order to obtain an upper bound to the charge costs and needs of the alternatives.

## 5 Correctness of the Abstract Interpretation

Let $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$, be the *context* function, which we assume well-typed according to the type system in [12]. Let us assume an execution of $e_f$ under some $E_0, h_0, k_0$ and $td_0$:

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

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).

The correctness argument is split into three parts. First, we shall define a notion of *correct signature* which formalises the intuition of the inferred $(\Delta, \mu, \sigma)$ being an upper bound of the actual $(\delta, m, s)$. Then we prove that the inference rules of Figure 2 are correct, assuming that all function applications are done to previously inferred functions, that the signatures given by $\Sigma$ for these functions are correct, and that the size analysis is correct. Finally, the correctness of the signature inference algorithm is proved, in particular when the function being inferred is recursive.

In order to define the notion of correct signature we have to give some previous definitions. We consider *region instantiations*, denoted by *Reg*, *Reg*$'$, …, which are partial mappings from region types $\rho$ to natural numbers $i$. Region instantiations are needed to specify the actual region $i$ to which every $\rho$ is instantiated at a given execution point. An instantiation *Reg* is *consistent* with a heap $h$, an environment $E$ and a type environment $\Gamma$ if *Reg* does not contradict the region instantiation obtained at runtime from $h$, $E$ and $\Gamma$, i.e. common type region variables are bound to the same actual region. A formal definition of consistency can be found in [12], where we also proved that if a function is well-typed, consistency of region instantiations is preserved along its execution.

**Definition 3.** *Given a pointer $p$ belonging to a heap $h$, the function size returns the number of cells in $h$ of the data structure starting at $p$:*

$$size(h[p \mapsto (j, C\ \overline{v_i}^n)], p) = 1 + \sum_{i \in RecPos(C)} size(h, v_i)$$

*where $RecPos(C)$ denotes the recursive positions of constructor $C$.*

For example, if $p$ points to the first cons cell of the list $[1, 2, 3]$ in the heap $h$ then $size(h, p) = 4$. We assume that $size(h, c) = 0$ for every heap $h$ and constant $c$.

**Definition 4.** *Given a sequence of sizes $\overline{s_i}^n$ for the input parameters, a number $k$ of regions and a region instantiation Reg, we say that*

- *$\Delta$ is an upper bound for $\delta$ in the context of $\overline{s_i}^n$, $k$ and Reg, denoted by $\Delta \succeq_{\overline{s_i}^n,k,Reg} \delta$ iff $\forall j \in \{0 \ldots k\} : \sum_{Reg\ \rho=j} \ \Delta\ \rho\ \overline{s_i}^n \geq \delta\ j$;*
- *$\mu$ is an upper bound for $m$, denoted $\mu \succeq_{\overline{s_i}^n} m$, iff $\mu\ \overline{s_i}^n \geq m$; and*
- *$\sigma$ is an upper bound for $s$, denoted $\sigma \succeq_{\overline{s_i}^n} s$, iff $\sigma\ \overline{s_i}^n \geq s$.*

A signature $(\Delta_g, \mu_g, \sigma_g)$ for a function $g$ is said to be *correct* if the components $(\Delta_g, \mu_g, \sigma_g)$ are upper bounds to the actual $(\delta, m, s)$ obtained from any execution of $g$. The correctness of the abstract interpretation rules in Fig. 2 can be proven provided the type signatures in $\Sigma$ are correct. Both the formal statement of this fact and the definition of correct signature can be found in [14].

In order to prove the correctness of the algorithms shown in the following section for recursive functions we need the abstract interpretation to be monotonic with respect to function signatures.

**Lemma 1.** *Let $f$ be a context function. Given $\Sigma_1, \Sigma_2$, $\Gamma$, and td such that $\Sigma_1 \sqsubseteq \Sigma_2$, then $[\![e]\!]\ \Sigma_1\ \Gamma\ td \sqsubseteq [\![e]\!]\ \Sigma_2\ \Gamma\ td$.*

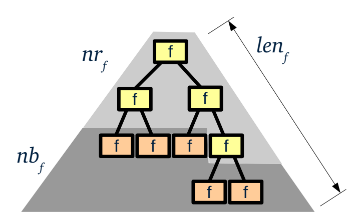
*Proof.* By structural induction on $e$, because $+$ and $\sqcup$ are monotonic. □

## 6 Space Inference Algorithms

Given a recursive function $f$ with $n+m$ arguments, the algorithms for inferring $\Delta_f$ and $\sigma_f$ do not depend on each other, while the algorithm for inferring $\mu_f$ needs a correct value for $\Delta_f$. We will assume that $\mu_f$, $\sigma_f$, and the cost functions in $\Delta_f$, do only depend on arguments of $f$ non-increasing in size. The consequence of this restriction is that the costs charged to regions, or to the stack, by the most external call to $f$ are safe upper bounds to the costs charged by all the lower level internal calls. This restriction holds for the majority of programs occurring in the literature. Of course, it is always possible to design an example where the charges grow as we progress towards the leafs of the call-tree.

We assume that, for every recursive function $f$, there has been an analysis giving the following information as functions of the argument sizes $\overline{x_i}^n$:

1. $nr_f$, an upper bound to the number of calls to $f$ invoking $f$ again. It corresponds to the internal nodes of $f$'s call tree.
2. $nb_f$, an upper bound to the number of *basic* calls to $f$. It corresponds to the leaves of $f$'s call tree.
3. $len_f$, an upper bound to the maximum length of $f$'s call chains. It corresponds to the height of $f$'s call tree.




In general, these functions are not independent of each other. For instance, with linear recursion we have $nr_f = len_f - 1$ and $nb_f = 1$. However, we will not assume a fixed relation between them. If this relation exists, it has been already used to

$$
\begin{array}{l}
\mathit{splitExp}_f\ [\![e]\!] = (e, \#) \qquad\quad \text{if } e = c, x, C\ \overline{a_i}^n\ @\ r,\ \text{or}\ g\ \overline{a_i}^n\ @\ \overline{r_j}^m \text{ with } g \neq f \\
\mathit{splitExp}_f\ [\![f\ \overline{a_i}^n\ @\ \overline{r_j}^m]\!] = (\#, f\ \overline{a_i}^n\ @\ \overline{r_j}^m) \\
\mathit{splitExp}_f\ [\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2]\!] = (e_b, e_r) \\
\qquad \mathbf{where}\ (e_{1b}, e_{1r}) = \mathit{splitExp}_f\ [\![e_1]\!] \\
\qquad\qquad\quad (e_{2b}, e_{2r}) = \mathit{splitExp}_f\ [\![e_2]\!] \\
\qquad\qquad\quad e_b = \begin{cases} \# & \text{if } e_{1b} = \# \text{ or } e_{2b} = \# \\ \mathbf{let}\ x_1 = e_{1b}\ \mathbf{in}\ e_{2b} & \text{otherwise} \end{cases} \\
\qquad\qquad\quad e_r = \begin{cases} \# & \text{if } e_{1r} = \# \text{ and } e_{2r} = \# \\ \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_{2r} & \text{if } e_{1r} = \# \text{ and } e_{2r} \neq \# \\ \mathbf{let}\ x_1 = e_{1r}\ \mathbf{in}\ e_2 & \text{if } e_{1r} \neq \# \text{ and } e_{2r} = \# \\ \bigsqcup \left\{ \begin{array}{l} \mathbf{let}\ x_1 = e_{1b}\ \mathbf{in}\ e_{2r} \\ \mathbf{let}\ x_1 = e_{1r}\ \mathbf{in}\ e_2 \end{array} \right\} & \text{otherwise} \end{cases} \\
\mathit{splitExp}_f\ [\![\mathbf{case}(!)\ x\ \mathbf{of}\ \overline{alt_i}^n]\!] = (e_b, e_r) \\
\qquad \mathbf{where}\ (\overline{alt_{ib}}^n, \overline{alt_{ir}}^n) = \mathit{unzip}\ (\mathit{map}\ \mathit{splitAlt}_f\ \overline{alt_i}^n) \\
\qquad\qquad\quad e_b = \begin{cases} \# & \text{if } alt_{ib} = \# \to \# \text{ for all } i \in \{1 \ldots n\} \\ \mathbf{case}(!)\ x\ \mathbf{of}\ \overline{alt_{ib}}^n & \text{otherwise} \end{cases} \\
\qquad\qquad\quad e_r = \begin{cases} \# & \text{if } alt_{ir} = \# \to \# \text{ for all } i \in \{1 \ldots n\} \\ \mathbf{case}(!)\ x\ \mathbf{of}\ \overline{alt_{ir}}^n & \text{otherwise} \end{cases} \\
\mathit{splitAlt}_f\ [\![C\ \overline{x_j}^n \to e]\!] = (alt_b, alt_r) \\
\qquad \mathbf{where}\ (e_b, e_r) = \mathit{splitExp}_f\ e \\
\qquad\qquad\quad alt_b = \begin{cases} \# \to \# & \text{if } e_b = \# \\ C\ \overline{x_j}^n \to e_b & \text{otherwise} \end{cases} \\
\qquad\qquad\quad alt_r = \begin{cases} \# \to \# & \text{if } e_r = \# \\ C\ \overline{x_j}^n \to e_r & \text{otherwise} \end{cases}
\end{array}
$$

**Fig. 3.** Function splitting a *Core-Safe* expression into its base and recursive cases

compute them. We will only assume that each function is a correct upper bound to its corresponding runtime figure. As a running example, let us consider the `splitAt` definition in Fig. 5(a). We would assume $nr_{\texttt{splitAt}} = \lambda n\ x. \min\{n, x-1\}$, $nb_{\texttt{splitAt}} = \lambda n\ x.1$ and $len_{\texttt{splitAt}} = \lambda n\ x. \min\{n+1, x\}$.

### 6.1 Splitting *Core-Safe* expressions

In order to do a more precise analysis, we separately analyse the base and the recursive cases of a *Core-Safe* function definition. Fig. 3 describes the functions *splitExp* and *splitAlt* which, given a *Safe* expression return the part of its body contributing to the base cases and the part contributing to the recursive cases. We introduce an empty expression # in order not to lose the structure of the original one when some parts are removed. These empty expressions charge null costs to both the heap and the stack. Since it might be not possible to split a expression into a single pair with the base and recursive cases, we introduce expressions of the form $\sqcup\ e_i$, whose abstract interpretation is the least upper bound of the interpretations of the $e_i$. It will also be useful to define another function which splits a *Core-Safe* expression into those parts that execute before and including the last recursive call, and those executed after the last recursive call, In Fig. 4 we define such function, called $\mathit{splitBA}_f$. In Fig. 5 we show a *Full-Safe* definition for a function `splitAt` splitting a list, and the result of applying *splitExp* and *splitBA* to its *Core-Safe* version.

If $e_f$ is $f$'s body, in the following we will assume $(e_r, e_b) = \mathit{splitExp}_f[\![e_f]\!]$ and $(e_{bef}, e_{aft}) = (\bigsqcup_i\ e^i_{bef}, \bigsqcup_i\ e^i_{aft})$, where $[\overline{(e^i_{bef}, e^i_{aft})}^n] = \mathit{splitBA}_f[\![e_r]\!]$.

$$
\begin{array}{l}
\mathit{splitBA}_f\ [\![e]\!] = [\,] \qquad\qquad \text{if } e = \#, c, x, C\ \overline{a_i}^n\ @\ r,\ \text{ or } g\ \overline{a_i}^n\ @\ \overline{r_j}^m \text{ with } g \neq f \\
\mathit{splitBA}_f\ [\![\sqcup_{i=1}^n e_i]\!] = \mathit{concat}\,[\mathit{splitBA}\ e_i \mid i \in \{1 \dots n\}] \\
\mathit{splitBA}_f\ [\![f\ \overline{a_i}^n\ @\ \overline{r_j}^m]\!] = [(f\ \overline{a_i}^n\ @\ \overline{r_j}^m, \#)] \\
\mathit{splitBA}_f\ [\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2]\!] = A +\!\!+\ B \\
\qquad \mathbf{where}\ (e_{1b}, e_{1r}) = \mathit{splitExp}_f\ [\![e_1]\!] \\
\qquad\qquad (e_{2b}, e_{2r}) = \mathit{splitExp}_f\ [\![e_2]\!] \\
\qquad\qquad e_{1r,split} = \mathit{splitBA}\ [\![e_{1r}]\!] \\
\qquad\qquad e_{2r,split} = \mathit{splitBA}\ [\![e_{2r}]\!] \\
\qquad\qquad A = [(\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_{2r,b}, \\
\qquad\qquad\qquad \mathbf{let}\ x_1 = \#\ \mathbf{in}\ e_{2r,a}) \mid (e_{2r,b}, e_{2r,a})\ \in\ e_{2r,split}] \\
\qquad\qquad B = \begin{cases} [\,] & \text{if } e_{2b} = \# \\ [(\mathbf{let}\ x_1 = e_{1r,b}\ \mathbf{in}\ \#, & \\ \qquad \mathbf{let}\ x_1 = e_{1r,a}\ \mathbf{in}\ e_{2b}) \mid (e_{1r,b}, e_{1r,a})\ \in\ e_{1r,split}] & \text{otherwise} \end{cases} \\
\mathit{splitBA}_f\ [\![\mathbf{case}(!)\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n]\!] = \\
\qquad\qquad \big[\big(\mathbf{case}(!)\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_{i,b}}^n, \mathbf{case}(!)\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_{i,a}}^n\big) \\
\qquad\qquad\quad \mid (e_{1,b}, e_{1,a}) \in \mathit{splitBA}_f\ [\![e_1]\!], \dots, (e_{n,b}, e_{n,a}) \in \mathit{splitBA}_f\ [\![e_n]\!]\big]
\end{array}
$$

**Fig. 4.** Function splitting a *Core-Safe* expression into its parts executing before and after the last recursive call

### 6.2 Algorithm for computing $\boldsymbol{\Delta_f}$

The idea here is to separately compute the charges to regions of the recursive and non-recursive parts of $f$'s body, and then multiply these charges by respectively the number of internal and leaf nodes of $f$'s call-tree.

1. Set $\Sigma\ f = ([\,]_f, 0, 0)$.
2. Let $(\Delta_r, \_, \_) = [\![e_r]\!]\ \Sigma\ \Gamma\ (n+m)$
3. Let $(\Delta_b, \_, \_) = [\![e_b]\!]\ \Sigma\ \Gamma\ (n+m)$
4. Then, $\Delta_f \stackrel{\text{def}}{=} \Delta_r\mid_{\rho\neq\rho_{self}} \times nr_f + \Delta_b\mid_{\rho\neq\rho_{self}} \times nb_f$.

If we apply the abstract interpretation rules for the base cases of our `splitAt` example in Fig. 5(b) we get $\Delta_b = [\rho \mapsto \lambda n\ x.1 \mid \rho \in \{\rho_1, \rho_2, \rho_3\}]$. If we apply them to the recursive case in Fig. 5(d) we get $\Delta_r = [\rho \mapsto \lambda n\ x.1 \mid \rho \in \{\rho_1, \rho_2\}]$. The resulting $\Delta_{\texttt{splitAt}}$ is shown in Fig. 7.

**Lemma 2.** *If $nr_f, nb_f$, and all the size functions belong to $\mathbb{F}$, then all functions in $\Delta_f$ belong to $\mathbb{F}$.*

**Lemma 3.** *$\Delta_f$ is a correct abstract heap for $f$.*

*Proof.* This is a consequence of $nr_f$, $nb_f$, and all the size functions being upper bounds of their respective runtime figures, and of $\Delta_r$, $\Delta_b$ being upper bounds of respectively the $f$'s call-tree internal and leaf nodes heap charges. □

Let us call $\mathbb{I}_\Delta : \mathbb{D} \to \mathbb{D}$ to an iteration of the interpretation function, i.e. $\mathbb{I}_\Delta(\Delta_1) = \Delta_2$, being $\Delta_2$ the abstract heap obtained by initially setting $\Sigma\ f = (\Delta_1, 0, 0)$, then computing $(\Delta, \_, \_) = [\![e_r]\!]\ \Sigma\ \Gamma\ (n+m)$, and then defining $\Delta_2 = \Delta\mid_{\rho\neq\rho_{self}}$.

**Lemma 4.** *For all $n$, $\mathbb{I}^n_\Delta(\Delta_f)$ is a correct abstract heap for $f$.*

```
splitAt 0 xs     = ([],xs)
splitAt n []     = ([],[])
splitAt n (x:xs) = (x:xs1,xs2)
   where (xs1,xs2) = split (n-1) xs
```

(a) *Full-Safe* version

```
splitAt n xs @ r1 r2 r3 =
 case n of
  _ -> case xs of
       (: y1 y2) ->
          let y3 = let x6 = - n 1 in
                   splitAt x6 y2 @ r1 r2 r3 in #
```

(b) *Core-Safe* up to the last call

```
splitAt n xs @ r1 r2 r3 =
 case n of
  0 -> let  x1 = [] @ r2 in
       let  x2 = (x1,xs) @ r3 in x2
  _ -> case xs of
        [] -> let x4 = [] @ r2 in
              let x3 = [] @ r1 in
              let x5 = (x4,x3) @ r3 in x5
```

(c) *Core-Safe* base cases

```
splitAt n xs @ r1 r2 r3 =
 case n of
  _ -> case xs of
       (: y1 y2) ->
          let y3  = let x6 = - n 1 in
                    splitAt x6 y2 @ r1 r2 r3 in
          let xs1 = case y3 of (y4,y5) -> y4 in
          let xs2 = case y3 of (y6,y7) -> y7 in
          let x7  = (: y1 xs1) @ r2 in
          let x8  = (x7,xs2) @ r3 in x8
```

(d) *Core-Safe* recursive cases

```
splitAt n xs @ r1 r2 r3 =
 case n of
  _ -> case xs of
       (: y1 y2) ->
          let y3  = # in
          let xs1 = case y3 of (y4,y5) -> y4 in
          let xs2 = case y3 of (y6,y7) -> y7 in
          let x7  = (: y1 xs1) @ r2 in
          let x8  = (x7,xs2) @ r3 in x8
```

(e) *Core-Safe* after the last call

**Fig. 5.** Splitting a *Core-Safe* definition

*Proof.* This is a consequence of $\mathbb{D}$ being a complete lattice, $\mathbb{I}_\Delta$ being monotonic in $\mathbb{D}$, and $\mathbb{I}_\Delta(\Delta_f) \sqsubseteq \Delta_f$. As $\mathbb{I}_\Delta$ is reductive at $\Delta_f$ then, by Tarski's fixpoint theorem, $\mathbb{I}^n_\Delta(\Delta_f)$ is above the least fixpoint of $\mathbb{I}_\Delta$ for all $n$. $\square$

As the algorithm for $\mu_f$ critically depends on how good is the result for $\Delta_f$, it is advisable to spend some time iterating the interpretation $\mathbb{I}_\Delta$ in order to get better results for $\mu_f$.

### 6.3 Algorithm for computing $\mu_f$

We separately infer the part $\mu_{self}$ of $\mu_f$ due to space charges to the *self* region of $f$. As the *self* regions for $f$ are stacked, this part only depends on the longest $f$'s call chain, i.e. on the height of the call-tree.

1. Set $\Sigma\ f = ([\,]_f, 0, 0)$.
2. Let $(\_, \mu_b, \_) = [\![e_b]\!]\ \Sigma\ \Gamma\ (n+m)$, i.e. the heap needs of the non-recursive part of $f's$ body.
3. Let $([\rho_{self} \mapsto \mu_{self}], \_, \_) = [\![e_{bef}]\!]\ \Sigma\ \Gamma\ (n+m)$, i.e. the charges to $\rho_{self}$ made by the part of $f's$ body before (and including) the last recursive call.
4. Let $(\_, \mu_{bef}, \_) = ([\![e_{bef}]\!]\ \Sigma\ \Gamma\ (n+m))|_{\rho \neq \rho_{self}}$, i.e. the heap needs of $f's$ body before the last recursive call, without considering the *self* region.
5. Let $(\_, \mu_{aft}, \_) = [\![e_{aft}]\!]\ \Sigma\ \Gamma\ (n+m)$, i.e. the heap needs of $f's$ body after the last recursive call.
6. Then, $\mu_f \stackrel{\text{def}}{=} |\,\Delta_f\,| + \mu_{self} \times (len_f - 1) + \sqcup\{\mu_{bef}, \mu_b, \mu_{aft}\}$.

The intuitive idea is that the charges to regions other than *self* are considered from the last but one call to $f$ of the longest chain call.

In our example, if we take as $e_b, e_{bef}$ and $e_{aft}$ the definitions of Fig. 5, we get $\mu_{self} = 0$, $\mu_b = 3$, $\mu_{bef} = 0$, and $\mu_{aft} = 2$. Hence $\mu_f = \lambda n\ x.2\min(n, x-1) + 6$.

**Lemma 5.** *If the functions in $\Delta_f$, $len_f$, and the size functions belong to $\mathbb{F}$, then $\mu_f$ belongs to $\mathbb{F}$.*

**Lemma 6.** *$\mu_f$ is a safe upper bound for $f$'s heap needs.*

*Proof.* This is a consequence of the correctness of the abstract interpretation rules, and of $\Delta_f, len_f$, and the size functions being upper bounds of their respective runtime figures. □

As in the case of $\Delta_f$, we can define an interpretation $\mathbb{I}_\mu$ taking any upper bound $\mu_1$ as input, and producing a better one $\mu_2 = \mathbb{I}_\mu(\mu_1)$ as output.

**Lemma 7.** *For all $n$, $\mathbb{I}^n_\mu(\mu_f)$ is a safe upper bound for $f$'s heap needs.*

*Proof.* This is a consequence of $\mathbb{F}$ being a complete lattice, $\mathbb{I}_\mu$ being monotonic in $\mathbb{F}$, and $\mathbb{I}_\mu$ being reductive at $\mu_f$. □

### 6.4 Algorithm for computing $\sigma_f$

The algorithm for inferring $\mu_f$ traverses $f$'s body from left to right because the abstract interpretation rules for $\mu$ need the charges to the previous heaps. For inferring $\sigma_f$ we can do it better because its rules are symmetrical. The main idea is to count only once the stack needs due to calling to external functions.

1. Let $(\_, \_, \sigma_b) = [\![e_b]\!]\ \Sigma\ \Gamma\ (n+m)$.
2. Let $(\_, \_, \sigma_{bef}) = [\![e_{bef}]\!]\ \Sigma[f \mapsto (\_, \_, \sigma_b)]\ \Gamma\ (n+m)$, i.e. the stack needs before the last recursive call, assuming as $f$'s stack needs those of the base case. This amounts to accumulating the cost of a leaf to the cost of an internal node of $f$'s call tree.
3. Let $(\_, \_, \sigma_{aft}) = [\![e_{aft}]\!]\ \Sigma\ \Gamma\ (n+m)$.
4. We define the following function $\mathcal{S}$ returning a natural number. Intuitively it computes an upper bound to the difference in words between the initial stack in a call to $f$ and the stack just before $e_{bef}$ is about to jump to $f$ again:

$$\begin{array}{ll}
\mathcal{S}\ [\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ \#]\!]\ td & = 2 + \mathcal{S}\ [\![e_1]\!]\ 0 \\
\mathcal{S}\ [\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2]\!]\ td & = \begin{cases} 1 + \mathcal{S}\ [\![e_2]\!]\ (td+1) & \text{if } f \notin e_1 \\ \sqcup\{2 + \mathcal{S}\ [\![e_1]\!]\ 0, 1 + \mathcal{S}\ [\![e_2]\!]\ (td+1)\} & \text{if } f \in e_1 \end{cases} \\
\mathcal{S}\ [\![\mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}]\!]\ td & = \bigsqcup_{r=1}^{n}(n_r + \mathcal{S}\ [\![e_r]\!]\ (td + n_r)) \\
\mathcal{S}\ [\![g\ \overline{a_i}^p\ @\ \overline{r_j}^q]\!]\ td & = p + q - td \\
\mathcal{S}\ [\![e]\!]\ td & = 0 \qquad \text{otherwise}
\end{array}$$

5. Then, $\sigma_f = (\mathcal{S}\ [\![e_{bef}]\!]\ (n+m)) * \sqcup\{0, len_f - 2\} + \sqcup\{\sigma_{bef}, \sigma_{aft}, \sigma_b\}$

In our example, if we denote by $e^{\mathtt{splitAt}}_{bef}$ the definition of Fig. 5(b) we get $\mathcal{S}\ [\![e^{\mathtt{splitAt}}_{bef}]\!]\ (2+3) = 9$ and, by applying the abstract interpretation rules to the definitions in Fig. 5(c),(b) and (e) we obtain $\sigma_b = \lambda n\ x.4$, $\sigma_{bef} = \lambda n\ x.13$ and $\sigma_{aft} = \lambda n\ x.9$. Hence $\sigma_f = 9\min\{n-1, x-2\} + 13 = 9\min\{n, x-1\} + 4$.

```
length []     = 0
length (x:xs) = 1 + length xs
```

$$
\begin{array}{ll}
\mathit{splitAt} & :: \mathit{Int} \to [a]@\rho_1 \to \rho_1 \to \rho_2 \to \rho_3 \to ([a]@\rho_2, [a]@\rho_1)@\rho_3 \\
\mathit{length} & :: [a]@\rho_1 \to \mathit{Int} \\
\mathit{merge} & :: [a]@\rho_1 \to [a]@\rho_1 \to \rho_1 \to [a]@\rho_1 \\
\mathit{msort} & :: [a]@\rho_1 \to \rho_1 \to \rho_2 \to [a]@\rho_2
\end{array}
$$

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

```
msort []     = []
msort (x:[]) = x:[]
msort xs     = merge (msort xs1) (msort xs2)
                 where (xs1,xs2) = splitAt (length xs / 2) xs
```

**Fig. 6.** *Full-Safe* mergesort program

| **Function** | **Heap charges $\Delta$** | **Heap needs $\mu$** | **Stack needs $\sigma$** |
|---|---|---|---|
| $\mathit{length}(x)$ | $\boxed{[\,]}$ | $\boxed{0}$ | $\boxed{5x-4}$ |
| $\mathit{splitAt}(n,x)$ | $\begin{bmatrix} \rho_1 \mapsto \boxed{1} \\ \rho_2 \mapsto \boxed{\min(n,x-1)+1} \\ \rho_3 \mapsto \boxed{\min(n,x-1)+1} \end{bmatrix}$ | $\boxed{2\min(n,x-1)+6}$ | $\boxed{9\min(n,x-1)+4}$ |
| $\mathit{merge}(x,y)$ | $\begin{bmatrix} \rho_1 \mapsto \boxed{\max(1,2x+2y-5)} \end{bmatrix}$ | $\boxed{\max(1,2x+2y-5)}$ | $\boxed{11(x+y-4)+20}$ |
| $\mathit{msort}^1(x)$ | $\begin{bmatrix} \rho_1 \mapsto \frac{x^2}{2} - \frac{1}{2} \\ \rho_2 \mapsto 2x^2 - 3x + 3 \end{bmatrix}$ | $0.31x^2 + 0.25x\log(x+1) + 14.3x + 0.75\log(x+1) + 10.3$ | $\max(80, 13x-10)$ |
| $\mathit{msort}^2(x)$ | $\begin{bmatrix} \rho_1 \mapsto \frac{x^2}{4} + x - \frac{1}{4} \\ \rho_2 \mapsto x^2 + x + 1 \end{bmatrix}$ | $\boxed{0.31x^2 + 8.38x + 13.31}$ | $\boxed{\max(80, 11x-25)}$ |
| $\mathit{msort}^3(x)$ | $\begin{bmatrix} \rho_1 \mapsto \frac{x^2}{8} + \frac{7x}{4} + \frac{9}{8} \\ \rho_2 \mapsto \frac{x^2}{2} + 4x + \frac{1}{2} \end{bmatrix}$ | $\boxed{0.31x^2 + 8.38x + 13.31}$ | $\boxed{\max(80, 11x-25)}$ |

**Fig. 7.** Cost results for the mergesort program

**Lemma 8.** *If $len_f$, and all the size functions belong to $\mathbb{F}$, then $\sigma_f$ belongs to $\mathbb{F}$.*

**Lemma 9.** *$\sigma_f$ is a safe upper bound for $f$'s stack needs.*

*Proof.* This is a consequence of the correctness of the abstract interpretation rules, and of $len_f$ being an upper bound to $f$'s call-tree height. □

Also in this case, it makes sense iterating the interpretation as we did with $\Delta_f$ and $\mu_f$, since it holds that $\mathbb{I}_\sigma(\sigma_f) \sqsubseteq \sigma_f$.

## 7 Case Studies

In Fig. 6 we show a *Full-Safe* version of the mergesort algorithm (the code for `splitAt` was presented in Fig. 5) with the types inferred by the compiler. Region $\rho_1$ is used inside `msort` for the internal call `splitAt n' xs @ r1 r1 self`, while region $\rho_2$ receives the charges made by `merge`. Notice that some charges to `msort`'s *self* region are made by `splitAt`. In Fig. 7 we show the results of our interpretation for this program as functions of the argument sizes. Remember that the size of a list (the number of its cells) is the list length plus one. The functions shown have been simplified with the help of a computer algebra tool. We show the fixpoints framed in grey. The upper bounds obtained for `length`, `splitAt`, and `merge` are exact and they are, as expected, fixpoints of the inference algorithm. For `msort` we show three iterations for $\Delta$ and $\sigma$, and another three for $\mu$ by using

| Function | Heap needs $\mu$ | Stack needs $\sigma$ |
|---|---|---|
| $partition(p, x)$ | $3x - 1$ | $9x - 5$ |
| $append(x, y)$ | $x - 1$ | $\max(8, 7x - 6)$ |
| $quicksort(x)$ | $3x^2 - 20x + 76$ | $\max(40, 20x - 27)$ |
| $insertD(e, x)$ | $1$ | $9x - 1$ |
| $insertTD(x, t)$ | $2$ | $\frac{11}{2}t + \frac{7}{2}$ |
| $fib(n)$ | $2^n + 2^{n-3} + 2^{n-4} - 3$ | $\max(10, 7n - 11)$ |
| $sum(n)$ | $0$ | $3n + 6$ |
| $sumT(a, n)$ | $0$ | $5$ |

**Fig. 8.** Cost results for miscellaneous *Safe* functions

```
sum 0 = 0
sum n = n + sum (n - 1)

sumT acc 0 = acc
sumT acc n = sumT (acc + n) (n - 1)
```

```
insertTD x Empty! = Node (Empty) x (Empty)
insertTD x (Node lt y rt)!
  | x == y = Node lt! y rt!
  | x > y  = Node lt! y (insertTD x rt)
  | x < y  = Node (insertTD x lt) y rt!
```

**Fig. 9.** Two summation functions and a destructive tree insertion function

the last $\Delta$. The upper bounds for $\Delta$ and $\mu$ are clearly over-approximated, since a term in $x^2$ arises which is beyond the actual space complexity class $O(x \log x)$ of this function. Let us note that the quadratic term's coefficient quickly decreases at each iteration in the inference of $\Delta$. Also, $\mu$ and $\sigma$ decrease in the second iteration but not in the third. This confirms the predictions of lemmas 4 and 7.

We have tried some more examples and the results inferred for $\mu$ and $\sigma$ after a maximum of three iterations are shown in Fig. 8, where the fixpoints are also framed in grey. There is a `quicksort` function using two auxiliary functions `partition` and `append` respectively classifying the list elements into those lower (or equal) and greater than the pivot, and appending two lists. We also show the destructive `insertD` function of Sec. 2, and a destructive version of the insertion in a search tree (its code is shown in Fig. 9). Both consume constant heap space. The next one shown is the usual Fibonacci function with exponential time cost, and using a constructed integer in order to show that an exponential heap space is inferred. Finally, we show two simple summation functions (its code also appears in Fig. 9), the first one being non-tail recursive, and the second being tail-recursive. Our abstract machine consumes constant stack space in the second case (see [11]). It can be seen that our stack inference algorithm is able to detect this fact.

## 8 Related and Future Work

Hughes and Pareto developed in [7] a type system and a type-checking algorithm which guarantees safe memory upper bounds in a region-based first-order functional language. Unfortunately, the approach requires the programmer to provide detailed consumption annotations, and it is limited to linear bounds. Hofmann and Jost's work [6] presents a type system and a type inference algorithm which, in case of success, guarantees linear heap upper bounds for a first-order functional language, and it does not require programmer annotations.

The national project AHA [16] aims at inferring amortised costs for heap space by using a variant of sized-types [8] in which the annotations are polynomials of any degree. They address two novel problems: polynomials are not necessarily monotonic and they are *exact* bounds, as opposed to approximate upper bounds. Type-checking is undecidable in this system and in [17, 15] they propose an inference algorithm for a list-based functional language with severe restrictions in which a combination of testing and type-checking is done. The algorithm does not terminate if the input-output size relation is not polynomial.

In [2], the authors directly analyse Java bytecode and compute safe upper bounds for the heap allocation made by a program. The approach uses the results of [1], and consists of combining a code transformation to an intermediate representation, a cost relations inference step, and a cost relations solving step. The second one combines ranking functions inference and partial evaluation. The results are impressive and go far beyond linear bounds. The authors claim to deal with data structures such as lists and trees, as well as arrays. Two drawbacks compared to our results are that the second step performs a global program analysis (so, it lacks modularity), and that only the allocated memory (as opposed to the live memory) is analysed. Very recently [3] they have added an escape analysis to each method in order to infer live memory upper bounds. The new results are very promising.

The strengths of our approach can be summarised as follows: (a) It scales well to large programs as each *Safe* function is separately inferred. The relevant information about the called functions is recorded in the signature environment; (b) We can deal with any user-defined algebraic datatype. Most of other approaches are limited to lists; (c) We get upper bounds for the *live* memory, as the inference algorithms take into account the deallocation of dead regions made at function termination; (d) We can get bounds of virtually any complexity class; and (e) It is to our knowledge the only approach in which the upper bounds can be easily improved just by iterating the inference algorithm.

The weak points that still require more work are the restrictions we have imposed to our functions: they must be non-negative and monotonic. This exclude some interesting functions such as those that destroy more memory than they consume, or those whose output size decreases as the input size increases. Another limitation is that the arguments of recursive *Safe* functions related to heap or stack consumption must be non-increasing. This limitation could be removed in the future by an analysis similar to that done in [1] in which they maximise the argument sizes across a call-tree by using linear programming tools. Of course, this could only be done if the size relations are linear.

Another open problem is inferring *Safe* functions with region-polymorphic recursion. Our region inference algorithm [13] frequently infers such functions, where the regions used in an internal call may differ from those used in the external one. This feature is very convenient for maximising garbage (i.e. allocations to the *self* region) but it makes more difficult the attribution of costs to regions.

## References


1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis Symposium, SAS'08*, pages 221–237. LNCS 5079, Springer, 2008.
2. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *Proc. Int. Symp. on Memory Management, ISMM'07, Montreal, Canada*, pages 105–116. ACM, 2007.
3. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *Proc. Int. Symp. on Memory Management, ISMM'09, Dublin, Ireland*, pages 129–138. ACM, 2009.
4. J. de Dios and R. Peña. A Certified Implementation on top of the Java Virtual Machine. In *Formal Method in Industrial Critical Systems, FMICS'09, Eindhoven (The Netherlands)*, pages 1–16. LNCS (to appear), Springer, November 2009.
5. J. de Dios and R. Peña. Formal Certification of a Resource-Aware Language Implementation. In *Int. Conf. on Theorem Proving in Higher Order Logics, TPHOL'09, Munich (Germany)*, pages 1–15. LNCS 5674 (to appear), Springer, August 2009.
6. 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.
7. R. J. M. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space; Towards Embedded ML Programming. In *Proc. Int. Conf. on Functional Programming, ICFP'99, Paris*, pages 70–81. ACM Press, Sept. 1999.
8. R. J. M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Conference Record of POPL '96: The* $23^{\mathrm{rd}}$ *ACM SIGPLAN-SIGACT*, pages 410–423, 1996.
9. T. Lindholm and F. Yellin. *The Java Virtual Machine Sepecification Second Edition.* The Java Series. Addison-Wesley, 1999.
10. 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.
11. M. Montenegro, R. Peña, and C. Segura. A Resource-Aware Semantics and Abstract Machine for a Functional Language with Explicit Deallocation. In *Workshop on Functional and (Constraint) Logic Programming, WFLP'08, Siena, Italy July, 2008 (to appear in ENTCS)*, pages 47–61, 2008.
12. 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.
13. M. Montenegro, R. Peña, and C. Segura. A simple region inference algorithm for a first-order functional language. In S. Escobar, editor, *Int. Work. on Functional and (Constraint) Logic Programming, WFLP 2009, Brasilia*, pages 63–77, 2009.
14. M. Montenegro, R. Peña, and C. Segura. A space consumption analysis by abstract interpretation (extended version), 2009. Available at `http://dalila.sip.ucm.es/safe`.
15. A. Tamalet, O. Shkaravska, and M. van Eekelen. Size Analysis of Algebraic Data Types. In Peter Achten, Pieter Koopman, and Marco T. Morazán, editors, *Trends in Functional Programming Volume 9 (TFP'08)*, pages 33–48. Intellect, 2009.
16. M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetsers. AHA: Amortized Space Usage Analysis. In *Selected Papers Trends in Functional Programming, TFP'07, New York*, pages 36–53. Intellect, 2008.
17. R. van Kesteren, O. Shkaravska, and M. van Eekelen. Inferring static non-monotonically sized types through testing. In *Proc. Work. on Functional and (Constraint) Logic Programming, WFLP'07, Paris, France.* ENTCS, Elsevier, 2007.