A Space Consumption Analysis By Abstract Interpretation (extended version) *

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

November, 2009

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 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

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

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:

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

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

$$\frac{j \leq k \quad (h', p') = \mathit{copy}(h, p, j) \quad m = \mathit{size}(h, p)}{E[x \mapsto p, r \mapsto j] \vdash h, k, td, x@r \Downarrow h', k, p', ([j \mapsto m], m, 2)} \; [\mathit{Var_2}]$$

$$\frac{\mathit{fresh}(q)}{E[x \mapsto p] \vdash h \uplus [p \mapsto w], k, td, x! \Downarrow h \uplus [q \mapsto w], k, q, ([\,], 0, 1)} \; [\mathit{Var_3}]$$

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

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

$$\frac{j \leq k \quad \mathit{fresh}(p) \quad E \cup [x_1 \mapsto p] \vdash h \uplus [p \mapsto (j, C \; \overline{v_i}^n)], k, td+1, e_2 \Downarrow h', k, v, (\delta, m, s)}{E[\overline{a_i \mapsto v_i}^n, r \mapsto j] \vdash h, k, td, \mathbf{let} \; x_1 = C \; \overline{a_i}^n @r \; \mathbf{in} \; e_2 \Downarrow h', k, v, (\delta + [j \mapsto 1], m+1, s+1)} \; [\mathit{Let_2}]$$

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

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

Figure 1: Resource-Aware Operational semantics of *Safe* expressions

```
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 the *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 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 (δ, 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 $\delta : \mathbb{N} \to \mathbb{Z}$ 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 $\mathit{dom}(\delta)$ we denote the subset of $\mathbb{N}$ in which δ is defined. By $|\delta|$ we mean the sum $\sum_{n \in \mathit{dom}(\delta)} \delta(n)$ 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:

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

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 $\eta : (\mathbb{R}^+ \cup \{+\infty\})^n \to \mathbb{R} \cup \{+\infty, -\infty\}$ 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 ρ. 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 Δ_f belongs to $\mathbb{D}$, and μ_f, σ_f belong to $\mathbb{F}$.*

The aim is that Δ_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 μ_f, σ_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 Σ be a global environment giving, for each *Safe* function g in scope, its signature $(\Delta_g, \mu_g, \sigma_g)$, let Γ 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 (Δ, μ, σ) 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 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

$$[\![c]\!]\ \Sigma\ \Gamma\ td = ([\,]_f, 0, 1) \quad [\mathit{Lit}]$$

$$[\![x]\!]\ \Sigma\ \Gamma\ td = ([\,]_f, 0, 1) \quad [\mathit{Var}]$$

$$\frac{\Gamma\ r = \rho \quad |x| = \eta}{[\![x\ @\ r]\!]\ \Sigma\ \Gamma\ td = ([\rho \mapsto \eta], \eta, 2)}\ [\mathit{Var_2}]$$

$$[\![x!]\!]\ \Sigma\ \Gamma\ td = ([\,]_f, 0, 1) \quad [\mathit{Var_3}]$$

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

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

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

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

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

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

Figure 2: Space inference rules for expressions with non-recursive applications

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, $\mathit{unify}\ \Gamma\ g\ \overline{a_i}^l\ \overline{r_j}^q$ computes a substitution θ 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 ρ_g is instantiated to the f's actual region type ρ_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* Γ*, a function* g *and the sequences* $\overline{a_i}^l$ *and* $\overline{r_j}^q$*, we say that* $\theta = \mathit{unify}\ \Gamma\ g\ \overline{a_i}^l\ \overline{r_j}^q$ *iff*

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

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:

$$\begin{array}{l}\Delta\ \rho^f_2 = \lambda\overline{x}^n.\Delta_g\ \rho^g_2\ \overline{(|a_i|\ \overline{x}^n)}^l \\ \Delta\ \rho^f_1 = \lambda\overline{x}^n.\Delta_g\ \rho^g_1\ \overline{(|a_i|\ \ \overline{x}^n)}^l + \Delta_g\ \rho^g_3\ \overline{(|a_i|\ \ \overline{x}^n)}^l + \Delta_g\ \rho^g_4\ \overline{(|a_i|\ \overline{x}^n)}^l + \Delta_g\ \rho^g_5\ \overline{(|a_i|\ \ \overline{x}^n)}^l\end{array}$$

Rules $[\mathit{Let_1}]$ and $[\mathit{Let_2}]$ reflect the corresponding resource-aware semantic rules in [11]. Rules $[\mathit{Case}]$ and $[\mathit{Case!}]$ use the least upper bound operators $\bigsqcup$ in order to obtain an upper bound to the charge costs and needs of the alternatives.

$$
\begin{array}{lcll}
build(h, c, B) & = & \emptyset & \\
build(h, p, T\ \overline{t_i}^n @ \overline{\rho_i}^m) & = & \emptyset & \text{if } p \notin dom(h) \\
build(h, p, T\ \overline{t_i}^n @ \overline{\rho_i}^m) & = & [\rho_m \to j] \cup \bigcup_{i=1}^{n_k} build(h, v_i, t_{ki}) & \text{if } p \in dom(h) \\
& \textbf{where} & h(p) = (j, C_k\ \overline{v_i}^{n_k}) & \\
& & \overline{t_{ki}}^{n_k} \to \rho_m \to T\ \overline{t_i}^n @ \overline{\rho_i}^m \trianglelefteq \Sigma(C_k) &
\end{array}
$$

Figure 3: Definition of *build* function.

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:

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

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).

The correctness argument is split into three parts. First, we shall define a notion of *correct signature* which formalises the intuition of the inferred (Δ, μ, σ) being an upper bound of the actual (δ, 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 Σ 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 ρ to natural numbers i. Region instantiations are needed to specify the actual region i to which every ρ is instantiated at a given execution point. An instantiation *Reg* is *consistent* with a heap h, an environment E and a type environment Γ if *Reg* does not contradict the region instantiation obtained at runtime from h, E and Γ, 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.

The function *build* (defined in Fig 3) follows the pointer chain of a given structure in order to construct a correspondence between region types and actual regions. The data structure is determined by the heap and the pointer given as first and second parameters; the third one is the type of the data structure.

Notice that the *build* function always return a region instantiation whose domain is a subset of the region type variables appearing in the type under consideration, that is, $dom\ build(h, v, t) \subseteq regions(t)$. However, there may exist region type variables in t which do not belong to the result of the resulting *build*. As an example, let us consider the following **data** declaration:

$$\textbf{data}\ EitherList\ a\ b\ @\ \rho_1\ \rho_2\ \rho_3 = Left\ ([a]@\rho_1)\ @\ \rho_3 \mid Right\ ([b]@\rho_2)\ @\ \rho_3$$

Under the heap $h = [p_1 \mapsto (2, Left\ p_2), p_2 \mapsto (1, [\,])]$ we get:

$$build(h, p_1, EitherList\ a\ b\ @\ \rho_5\ \rho_6\ \rho_7) = [\rho_5 \mapsto 1, \rho_7 \mapsto 2]$$

where the region type variable ρ_6 is not bound to any actual region.

It will be convenient to extend the notation of *build* to typing and value environments as follows:

$$build^*(h, E, \Gamma) = \bigcup_{\substack{x \in dom E \\ \neg regvar(x)}} build(h, E\ x, \Gamma\ x) \cup \bigcup_{\substack{r \in dom E \\ regvar(r)}} [\Gamma\ r \mapsto E\ r]$$

provided the result is well-defined, i.e. all occurring region instantiations are consistent with each other. This always holds, in particular, when the involved function is well-typed.

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. We shall define in a similar way the function $size^+$, which gives the number of cells of the whole DS pointed to by p.

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

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

- *Δ is an upper bound for δ in the context of $\overline{s_i}^n$, k and Reg, denoted by $\Delta \succeq_{\overline{s_i}^n, k, Reg} \delta$ iff*

$$\forall j \in \{0 \ldots k\} : \sum_{Reg\ \rho = j} \Delta\ \rho\ \overline{s_i}^n \geq \delta\ j$$

- *μ is an upper bound for m, denoted $\mu \succeq_{\overline{s_i}^n} m$, iff $\mu\ \overline{s_i}^n \geq m$; and*
- *σ 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 (δ, m, s) obtained from any execution of g. This is formalised in the following definition.

Definition 5 (Correct signature)**.** *Let $(\Delta_g, \mu_g, \sigma_g)$ the signature of a function definition $g\ \overline{y_i}^l\ @\ \overline{r'_j}^q = e_g$. This signature is said to be* correct *iff for all h, h', k, $\overline{v_i}^l$, $\overline{i_j}^q$, v, δ, m, s, Γ, t, $\overline{s_i}^n$ such that:*

1. *$E_g = [\overline{y_i \mapsto v_i}^l, \overline{r'_j \mapsto i_j}^q, self \mapsto k+1] \vdash h, k+1, l+q, e_g \Downarrow h', k+1, v, (\delta, m, s)$.*
2. *$\Gamma_g \vdash e_g : t$, according to the type system in [12].*
3. *$\forall i \in \{1 \ldots l\} : s_i = size(h, v_i)$*

then $\Delta_g \succeq_{\overline{s_i}^l, k, Reg} \delta|_k\ \wedge\ \mu_g \succeq_{\overline{s_i}^l} m\ \wedge\ \sigma_g \succeq_{\overline{s_i}^l} s$ for every region instantiation Reg consistent with h, E_g and Γ_g.

Definition 6 (Correct size analysis)**.** *Let f be the context function. The size analysis $|\cdot|$ is correct if for all subexpressions e of its body such that the judgement:*

$$E \vdash h, k_0, td, e \Downarrow h', k_0, v, (\delta, m, s)$$

belongs to the derivation in (1) it holds that

$$\forall x \in dom\ E : |x|\ \overline{s_i}^n \geq size(h, E\ x) \text{ where } s_i = size(h_0, E_0\ x_i) \text{ for each } i \in \{1 \ldots n\}$$

with E_0, h_0 and $\overline{x_i}^n$ being respectively the initial value environment, the initial heap and the input parameters corresponding to the context function.

The correctness of the abstract interpretation rules in Fig. 2 can be proven provided the type signatures in Σ are correct.

Lemma 1. *Let h be a fixed heap, t a nonfunctional type, and θ a region substitution such that $regions(t) \subseteq dom\ \theta$. For every pointer p belonging to the domain of h:*

$$dom(build(h, p, t)) \subseteq dom(build(h, p, \theta\ t) \circ \theta)$$
$$\forall \rho \in dom(build(h, p, t)) : build(h, p, t)\ \rho = build(h, p, \theta\ t)\ (\theta\ \rho)$$

provided both $build(h, p, t)$ and $build(h, p, \theta\ t)$ are well-defined.

Proof. By induction on $size^+(h,p)$.

If $size^+(h,p)=0$ then we get a contradiction, as t would be a basic type B or an algebraic type with $p \notin dom\ h$. Therefore, we shall assume in what follows that t is an algebraic type and $p \in dom\ h$.

Assuming that $t = T\ \overline{t_i}^l\ @\ \overline{\rho_j}^q$, $h(p) = (k, C\ \overline{v_i}^n)$, and that $\overline{t'_i}^n \to \rho'_m \to t$ is an instantiation of the data constructor C, we shall prove:

$$[\rho \mapsto j] \in build(h,p,t) \Rightarrow [\rho \mapsto j] \in build(h,p,\theta\ t) \circ \theta$$

Firstly, we know that $\rho \in dom\ \theta$, since $\rho \in dom\ (build(h,p,t)) \subseteq regions(t)$. We can unfold the definition of $build(h,p,t)$ in order to get:

$$build(h,p,t) = [\rho'_m \mapsto k] \cup \bigcup_{i=1}^{n} build(h, v_i, t'_i) \tag{2}$$

and hence:

$$build(h,p,\theta\ t) = [\theta\ \rho'_m \mapsto k] \cup \bigcup_{i=1}^{n} build(h, v_i, \theta\ t'_i) \tag{3}$$

On the one hand, if $\rho = \rho'_m$ then we get $j = k$ from (2) and it holds that $build(h,p,\theta\ t)\ (\theta\ \rho) = k = j$ from (3). Therefore, the binding $[\rho \mapsto j]$ belongs to the result of $build(h,p,\theta\ t) \circ \theta$. On the other hand, if we assume that $\rho \neq \rho'_m$ then for some $i \in \{1 \ldots n\}$:

$$\begin{array}{rcll}
[\rho \mapsto j] \in build(h,v_i,t'_i) & \Rightarrow & [\rho \mapsto j] \in build(h,v_i,\theta\ t'_i) \circ \theta & \{\text{by I.H.}\} \\
& \Rightarrow & [\theta\ \rho \mapsto j] \in build(h,v_i,\theta\ t'_i) & \\
& \Rightarrow & [\theta\ \rho \mapsto j] \in build(h,p,\theta\ t) & \\
& \Rightarrow & [\rho \mapsto j] \in build(h,p,\theta\ t) \circ \theta &
\end{array}$$

□

Lemma 2. *Let f be the context function. Then, for every subexpression e of the body e_f of the context function and E, h, h', v such that $E \vdash h, k_0, e \Downarrow h', k_0, v$ belongs to the derivation (1), it holds that $\forall x \in dom\ E : size(h, E\ x) \geq size(h', E\ x)$.*

Proof. It is a property of the big-step semantics, which can be proven by simple inspection of the corresponding rules. □

Theorem 1 (Correctness of the type system)**.** *Let us assume that $E \vdash h, k, e \Downarrow h'', k, v, (\delta, m, s)$ and that $\Gamma \vdash e : t$. If $Reg = build^*(h, E, \Gamma)$ is well-defined then for every h', E' and Γ' occurring in these derivations, the region instantiation $build^*(h', E', \Gamma')$ is consistent with Reg and so is the result of $build(h, v, t)$.*

Proof. It follows from the correctness theorem in [12]. □

The following theorem establishes the correctness of the abstract interpretation for non-recursive functions.

Theorem 2. *Let f a non-recursive context function. For each subexpression e of e_f and E, Σ, Γ, td, Δ, μ, σ, h, ,h', v, ,t, δ, m and s such that:*

1. *Every function call $g\ \overline{a_i}^l\ @\ \overline{r'_j}^q$ in e satisfies $g \in dom\ \Sigma$ and $\Sigma(g)$ is correct*
2. *$[\![e]\!]\ \Sigma\ \Gamma\ td = (\Delta, \mu, \sigma)$, where every occurrence of $|x|$ in its derivation has been inferred with a correct size analysis.*
3. *$E \vdash h, k_0, td, e \Downarrow h', k_0, v, (\delta, m, s)$, belonging to (1)*
4. *$\Gamma \vdash e : t$, according to the type system in [12].*

then $\Delta \succeq_{\overline{s_i}^n, k_0, Reg} \delta$, $\mu \succeq_{\overline{s_i}^n} m$ and $\sigma \succeq_{\overline{s_i}^n} s$, where $s_i = size(h, E_0\ x_i)$ for each $i \in \{1 \ldots n\}$, and each region instantiation Reg consistent with $build^*(h, E, \Gamma)$ such that $dom\ Reg = dom\ \Delta$.

Proof. By structural induction on e. In the following we shall leave out the $\overline{s_i}^n$ and k_0 subscripts in the $\succeq$ relations for a better readability.

- **Cases** $e \equiv c$**,** $e \equiv x$ **and** $e \equiv x!$

  We get $\Delta = [\,]_f = \lambda\rho.\lambda\overline{x_i}^n.0$, $\mu = \lambda\overline{x_i}^n.0$ and $\sigma = \lambda\overline{x_i}^n.1$. We prove:

  1. $\Delta \succeq \delta$
     Since for every $i \in \{0 \ldots k_0\}$ we get:
     $$\sum_{Reg\ \rho = i} \Delta\ \rho\ \overline{s_i}^n = 0 = \delta\ i$$
  2. $\mu \succeq m$, since $\mu\ \overline{s_i}^n = 0 = m$
  3. $\sigma \succeq s$, since $\sigma\ \overline{s_i}^n = 1 = s$

- **Case** $e \equiv x@r$

  Let $m = size(h, E\ x)$. We prove:

  1. $\Delta \succeq \delta$. By rule $[Var_2]$ we get $|x| = \eta$, $\Gamma\ r = \rho$ and
     $$\Delta = \lambda\rho'.\begin{cases} \eta & \text{if } \rho' = \rho \\ \lambda\overline{x_i}^n.0 & \text{if } \rho' \neq \rho \end{cases}$$
     Let $i \in \{0 \ldots k_0\}$. Firstly we assume $i = E\ r$. Since $\Gamma\ r = \rho$ and *Reg* is consistent with $build^*(h, E, \Gamma)$, then $Reg\ \rho = i$. Therefore:
     $$\begin{array}{rcll} \displaystyle\sum_{Reg\ \rho' = E\ r} \Delta\ \rho'\ \overline{s_i}^n & = & \displaystyle\sum_{\substack{Reg\ \rho' = E\ r \\ \rho' \neq \rho}} \Delta\ \rho'\ \overline{s_i}^n + \Delta\ \rho\ \overline{s_i}^n & \\ & = & 0 + \Delta\ \rho\ \overline{s_i}^n & \\ & = & \eta\ \overline{s_i}^n & \\ & = & |x|\ \overline{s_i}^n & \\ & \geq & size(h, E\ x) & \{\text{by Definition 6}\} \\ & = & \delta\ (E\ r) & \end{array}$$
     For the remaining case, $i \neq E\ r$, every ρ' such that $Reg\ \rho' = i \neq E\ r$ must be distinct from ρ, as the consistency constraint of *Reg* forces $Reg\ \rho = E\ r$. Therefore:
     $$\sum_{Reg\ \rho' = i} \Delta\ \rho'\ \overline{s_i}^n = (\lambda\overline{x_i}^n.0)\ \overline{s_i}^n = 0 = \delta\ i$$
  2. $\mu \succeq m$, since:
     $$\mu\ \overline{s_i}^n = \eta\ \overline{s_i}^n = |x|\ \overline{s_i}^n \geq size(h, E\ x) = m$$
  3. $\sigma \succeq s$, since:
     $$\sigma\ \overline{s_i}^n = 2 = s$$

- **Case** $e \equiv$ **let** $x_1 = C\ \overline{a_i}^l@r$ **in** e_2

  Let us denote the extended environment and heap by E_1 and h_1:
  $$\begin{array}{rcll} E_1 & = & E \cup [x_1 \mapsto p] & \\ h_1 & = & h \uplus \left[p \mapsto (j, C\ (E\ \overline{a_i}^l)\right] & \quad \text{where } j = E\ r \end{array}$$
  By the corresponding rules we get:

$$\begin{array}{l} [\![e_2]\!]\ \Sigma\ \Gamma\ (td+1) = (\Delta_1, \mu_1, \sigma_1) \\ E_1 \vdash h_1, k_0, td+1, e_2 \Downarrow h', k_0, v, (\delta_1, m_1, s_1) \\ \Gamma_1 + [x_1 : \tau_1] \vdash e_2 : t \end{array}$$

for some Δ_1, μ_1, σ_1, δ_1, m_1, s_1, τ_1, t and Γ_1. By the rules of the type system, $\Gamma_1 \sqsubseteq \Gamma$. By applying the induction hypothesis we get $\Delta_1 \succeq_{\overline{s_i}^n, k_0, Reg'} \delta_1$, $\mu_1 \succeq m_1$ and $\sigma_1 \succeq s_1$, for every Reg' consistent with $build^*(h_1, E_1, \Gamma_1)$. In particular, by Theorem 1 the current Reg satisfies this condition.

1. $\Delta \succeq \delta$. Let $i \in \{0 \ldots k_0\}$ a region number. If $i = j$, where j is the region where the new cell is created, then $Reg\ \rho = j$, since $\Gamma\ r = \rho$, $E\ r = j$ and Reg is consistent with $[\Gamma\ r \mapsto E\ r] \in build^*(h, E, \Gamma)$. Hence:

$$\begin{array}{rcl} \displaystyle\sum_{Reg\ \rho'=j} \Delta\ \rho'\ \overline{s_i}^n & = & \displaystyle\sum_{\substack{Reg\ \rho'=j \\ \rho' \neq \rho}} (\Delta\ \rho'\ \overline{s_i}^n) + \Delta\ \rho\ \overline{s_i}^n \\ & = & \displaystyle\sum_{\substack{Reg\ \rho'=j \\ \rho' \neq \rho}} (\Delta_1\ \rho'\ \overline{s_i}^n) + \Delta_1\ \rho\ \overline{s_i}^n + 1 \\ & = & \displaystyle\sum_{Reg\ \rho'=j} (\Delta_1\ \rho'\ \overline{s_i}^n) + 1 \\ & \geq & (\delta_1\ j) + 1 \\ & = & \delta\ j \end{array}$$

On the other hand, if $i \neq j$ then for every ρ' such that $Reg\ \rho' = i$ it holds that $Reg\ \rho' \neq j$, which implies $\rho' \neq \rho$. Therefore:

$$\sum_{Reg\ \rho'=i} \Delta\ \rho'\ \overline{s_i}^n = \sum_{Reg\ \rho'=i} \Delta_1\ \rho'\ \overline{s_i}^n \geq \delta_1\ i = \delta\ i$$

2. $\mu \succeq m$. It follows trivially from the induction hypothesis:

$$\mu\ \overline{s_i}^n = \mu_1\ \overline{s_i}^n + 1 \geq m_1 + 1 = m$$

3. $\sigma \succeq s$. Similarly:

$$\sigma\ \overline{s_i}^n = \sigma_1\ \overline{s_i}^n + 1 \geq s_1 + 1 = s$$

- **Case** $e \equiv g\ \overline{a_i}^l\ @\ \overline{r'_j}^q$

We shall assume that $\Sigma\ g \equiv g\ \overline{y_i}^l\ @\ \overline{r''_j}^q = e_g$ and, by using the corresponding rule:

$$\begin{array}{c} E_g \vdash h, k_0+1, l+q, e_g \Downarrow h', k_0+1, v, (\delta_g, m_g, s_g) \\ \text{where } E_g = \left[\overline{y_i \mapsto E\ a_i}^l, \overline{r''_j \mapsto r'_j}^q, self \mapsto k_0+1\right] \end{array}$$

Moreover, we assume that the function g has already been inferred and that its signature $(\Delta_g, \mu_g, \sigma_g)$ is correct. This implies, on the one hand, that the function g is well-typed and if $\Gamma\ g = \forall \overline{\alpha}\ \overline{\rho}.\overline{t_i}^l \rightarrow \overline{\rho_j}^q \rightarrow t$ then we can build a typing environment $\Gamma_g = \Gamma' + [\overline{y_i : t_i}^l, \overline{r''_j : \rho_j}^q, self : \rho_{self}]$ such that $\Gamma_g \vdash e_g : t$. On the other hand, if $s_{i,g}$ denote the size of the i-th actual argument before evaluating the function's body (i.e. $\forall i \in \{1 \ldots l\} : s_{i,g} = size(h, E_g\ y_i)$) then:

$$\Delta_g \succeq_{\overline{s_{i,g}}^l, k_0, Reg'} \delta_g|_{k_0} \qquad \mu_g \succeq_{\overline{s_{i,g}}^l} m_g \qquad \sigma_g \succeq_{\overline{s_{i,g}}^l} s$$

for each Reg' consistent with $build^*(h, E_g, \Gamma_g)$. Now we prove:

1. $\Delta \succeq_{\overline{s_i}^n, k_0, Reg} \delta$. Let $i \in \{0 \ldots k_0\}$. By the definition of Δ:

$$\sum_{Reg\ \rho = i} \Delta\ \rho\ \overline{s_i}^n = \sum_{Reg\ \rho = i}\ \sum_{\theta\ \rho' = \rho} \Delta_g\ \rho'\ \overline{|a_i|\ \overline{s_i}^n}^l$$

where $\theta = \mathit{unify}\ \Gamma\ g\ \overline{a_i}^l\ \overline{r'_j}^q$. By Definition 6 we get for each $i \in \{1 \ldots l\}$

$$|a_i|\ \overline{s_i}^n \geq \mathit{size}(h, E\ a_i) = \mathit{size}(h, E_g\ y_i) = s_{i,g} \tag{4}$$

and hence, because of the monotonicity of $\Delta_g\ \rho$ for every ρ:

$$\sum_{Reg\ \rho = i} \Delta\ \rho\ \overline{s_i}^n \geq \sum_{Reg\ \rho = i}\ \sum_{\theta\ \rho' = \rho} \Delta_g\ \rho'\ \overline{s_{i,g}}^l = \sum_{(Reg \circ \theta)\ \rho' = i} \Delta_g\ \rho'\ \overline{s_{i,g}}^l$$

By definition of $\Delta_g \succeq_{h, k_0, (Reg \circ \theta)} \delta_g|_{k_0}$ and because of the fact that $i \neq k_0 + 1$, we can get the desired result:

$$\sum_{Reg\ \rho = i} \Delta\ \rho\ \overline{s_i}^n \geq \delta_g|_{k_0}\ i = \delta\ i$$

provided the involved region instantiation $(Reg \circ \theta)$ is consistent with $\mathit{build}^*(h, E_g, \Gamma_g)$. We shall prove this as follows: let us assume that $[\rho \mapsto k] \in Reg \circ \theta$ (which, in turn, implies that $[\theta\ \rho \mapsto k] \in Reg$) and that $[\rho \mapsto k'] \in \mathit{build}^*(h, E_g, \Gamma_g)$ for some k and k'. We show that $k = k'$:

   - If $[\rho \mapsto k'] \in \mathit{build}(h, E_g\ y_i, \Gamma_g\ y_i)$ for some $i \in \{1 \ldots l\}$ then, by Lemma 1 we would get:

   $$[\theta\ \rho \mapsto k'] \in \mathit{build}(h, E_g\ y_i, \theta(\Gamma_g\ y_i)) = \mathit{build}(h, E\ a_i, \Gamma\ a_i)$$

   with the last step justified by the definition of *unify*. However, in order to apply this Lemma we have to show that the involved region instantiations are well-defined. However, this follows trivially from Theorem 1, as $\mathit{build}(h, E_g\ y_i, \theta(\Gamma_g\ y_i)) = \mathit{build}(h, E\ a_i, \Gamma\ a_i) \subseteq \mathit{build}^*(h, E, \Gamma)$.
   Therefore $[\theta\ \rho \mapsto k'] \in \mathit{build}^*(h, E, \Gamma)$. Since Reg is consistent with $\mathit{build}^*(h, E, \Gamma)$ and $[\theta\ \rho \mapsto k] \in Reg$, it follows that $k = k'$.
   - If $[\rho \mapsto k'] = [\Gamma_g\ r''_j \mapsto E_g\ r''_j]$ for some $j \in \{1 \ldots q\}$ then we get $[\theta\ \rho \mapsto k'] = [\theta\ (\Gamma_g\ r''_j) \mapsto E_g\ r''_j] = [\Gamma\ a_i \mapsto E\ r'_j]$ and, by using the same reasoning as the previous case, $k = k'$.

2. $\mu \succeq m$. We get:

$$\begin{array}{rcll} \mu\ \overline{s_i}^n & = & \mu_g\ \overline{|a_i|\ \overline{s_i}^n}^l & \\ & \geq & \mu_g\ \overline{s_{i,g}}^l & \{\text{because of (4) and monotonicity of } \mu_g\} \\ & \geq & m_g & \{\text{since } \mu_g \succeq_{\overline{s_{i,g}}^l} m_g\} \\ & = & m & \end{array}$$

3. $\sigma \succeq s$. Similarly, for σ_g being monotonic:

$$\begin{array}{rcl} \sigma\ \overline{s_i}^n & = & \sqcup\ \{l + q, \sigma_g\ \overline{(|a_i|\ \overline{s_j}^n)}^l - td + l + q\} \\ & \geq & \sqcup\ \{l + q, \sigma_g\ \overline{s_{j,g}}^l - td + l + q\} \\ & \geq & \sqcup\ \{l + q, \sigma_g\ s_1 - td + l + q\} \\ & = & s \end{array}$$

- **Case** $e \equiv$ **let** $x_1 = e_1$ **in** e_2

Let us assume that, by the corresponding rules, we get $E \vdash h, k_0, 0, e_1 \Downarrow h_2, k_0, v_1, (\delta_1, m_1, s_1)$ and $[\![e_1]\!]\ \Sigma\ \Gamma_1\ 0 = (\Delta_1, \mu_1, \sigma_1)$ for some δ_1, m_1, s_1, Γ_1, Δ_1, μ_1 and σ_1. In this case the induction hypothesis can be applied on e_1, so as to get:

$$\Delta_1 \succeq_{\overline{s_i}^n, k_0, Reg'} \delta_1 \qquad \mu_1 \succeq_{\overline{s_i}^n} m_1 \qquad \sigma_1 \succeq_{\overline{s_i}^n} s_1$$

for every Reg' consistent with $build^*(h, E, \Gamma_1)$, being Γ_1 the type environment under which e_1 is typed in the derivation $\Gamma \vdash e : t$. The current Reg meets trivially these constraints, so we can assume:

$$\Delta_1 \succeq_{\overline{s_i}^n, k_0, Reg} \delta_1 \qquad \mu_1 \succeq_{\overline{s_i}^n} m_1 \qquad \sigma_1 \succeq_{\overline{s_i}^n} s_1 \tag{5}$$

Similarly, we apply the induction hypothesis on e_2, in order to prove:

$$\Delta_2 \succeq_{\overline{s_i}^n, k_0, Reg'} \delta_2 \qquad \mu_2 \succeq_{\overline{s_i}^n} m_2 \qquad \sigma_2 \succeq_{\overline{s_i}^n} s_2$$

for every Reg' consistent with $build(h, E_2, \Gamma_2)$, with Γ_2 being the typing environment typing e_2 in the derivation of $\Gamma \vdash e : t$. Again, by Theorem 1 we get:

$$\Delta_2 \succeq_{\overline{s_i}^n, k_0, Reg} \delta_2 \qquad \mu_2 \succeq_{\overline{s_i}^n} m_2 \qquad \sigma_2 \succeq_{\overline{s_i}^n} s_2 \tag{6}$$

Now the results in (5) and (6) are combined in order to get the desired result:

1. $\Delta \succeq \delta$. For each $i \in \{0 \dots k_0\}$:

$$\begin{array}{rcl} \displaystyle\sum_{Reg\ \rho = i} (\Delta_1 + \Delta_2)\ \rho\ \overline{s_i}^n & = & \displaystyle\sum_{Reg\ \rho = i} (\Delta_1\ \rho\ \overline{s_i}^n + \Delta_2\ \rho\ \overline{s_i}^n) \\ & = & \displaystyle\sum_{Reg\ \rho = i} (\Delta_1\ \rho\ \overline{s_i}^n) + \sum_{Reg\ \rho = i} (\Delta_2\ \rho\ \overline{s_i}^n) \\ & \geq & (\delta_1\ i) + (\delta_2\ i) \\ & = & \delta\ i \end{array}$$

2. $\mu \succeq m$. For every $\rho \in dom\ \Delta_1$ there exists an $i \in \{0 \dots k_0\}$ such that $Reg\ \rho = i$. This allows us to establish:

$$|\Delta_1|\ \overline{s_i}^n = \sum_{\rho \in dom\ \Delta_1} \Delta_1\ \rho\ \overline{s_i}^n = \sum_{i=0}^{k_0} \sum_{Reg\ \rho = i} \Delta_1\ \rho\ \overline{s_i}^n \geq \sum_{i=0}^{k_0} \delta_1\ i = |\delta_1|$$

   Therefore:

$$\begin{array}{rcl} \mu\ \overline{s_i}^n & = & \sqcup\{\mu_1\ \overline{s_i}^n, |\Delta_1|\ \overline{s_i}^n + \mu_2\ \overline{s_i}^n\} \\ & \geq & \sqcup\{m_1, |\delta_1| + m_2\} \\ & = & m \end{array}$$

3. $\sigma \succeq s$. It follows trivially from the induction hypothesis:

$$\sigma\ \overline{s_i}^n = \sqcup\{2 + \sigma_1\ \overline{s_i}^n, 1 + \sigma_2\ \overline{s_i}^n\} \geq \sqcup\{2 + s_1, 1 + s_2\} = s$$

- **Case** $e \equiv$ **case** x **of** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^l$

We shall assume that the r-th branch is executed, that is, $h\ (E\ x) = (j, C_r\ \overline{v_i}^{n_r})$ for some j, v_1, $\dots$, v_{n_r} and $r \in \{1 \dots l\}$. Therefore the following judgements hold:

$$\begin{array}{l} [\![e_r]\!]\ \Sigma\ \Gamma_r\ (td + n_r) = (\Delta_r, \mu_r, \sigma_r) \\ E_r \vdash h, k_0, td + n_r, e_r \Downarrow h', k_0, v, (\delta_r, m_r, s_r) \end{array}$$

for some Δ_r, μ_r, σ_r, δ_r, m_r, s_r and where E_r denote the extended environment:

$$E_r \quad = \quad E \cup [\overline{x_{rj} \mapsto v_j}^{n_r}]$$

From the induction hypothesis and Theorem 1 it follows that $\Delta_r \succeq_{h, k_0, Reg} \delta_r$, $\mu_r \succeq m_r$ and $\sigma_r \succeq s_r$, which allows us to prove:

1. $\Delta \succeq \delta$. Let $i \in \{0 \dots k_0\}$

$$\begin{array}{rcl}
\sum\limits_{Reg\ \rho=i} (\Delta\ \rho\ \overline{s_i}^n) & = & \sum\limits_{Reg\ \rho=i} \left(\left(\sqcup_{i=1}^{l} \Delta_i\right)\ \rho\ \overline{s_i}^n \right) \\
& = & \sum\limits_{Reg\ \rho=i} \max\{\Delta_i\ \rho\ \overline{s_i}^n \mid 1 \leq i \leq l\} \\
& \geq & \sum\limits_{Reg\ \rho=i} \Delta_r\ \rho\ \overline{s_i}^n \\
& \geq & \delta_r\ i \\
& = & \delta\ i
\end{array}$$

2. $\mu \succeq m$, since:

$$\begin{array}{rcl}
\mu\ \overline{s_i}^n & = & \sqcup_{i=1}^{l} \mu_i\ \overline{s_i}^n \\
& = & \max\{\mu_i\ \overline{s_i}^n \mid 1 \leq i \leq l\} \\
& \geq & \mu_r\ \overline{s_i}^n \\
& \geq & m_r \\
& = & m
\end{array}$$

3. $\sigma \succeq s$, since:

$$\begin{array}{rcl}
\sigma\ \overline{s_i}^n & = & \sqcup_{i=1}^{l} (\sigma_i + n_i)\ \overline{s_i}^n \\
& = & \max\{\sigma_i\ \overline{s_i}^n + n_i \mid 1 \leq i \leq l\} \\
& \geq & \sigma_r\ \overline{s_i}^n + n_r \\
& \geq & s_r + n_r \\
& = & s
\end{array}$$

- **Case** $e \equiv$ **case!** x **of** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^l$

Again, we assume that the r-th branch is executed. By denoting by E_r the extended environment, the following judgements follow from their respective rules:

$$\begin{array}{l}
[\![e_r]\!]\ \Sigma\ \Gamma_r\ (td + n_r) = (\Delta_r, \mu_r, \sigma_r) \\
E_r \vdash h_r, k_0, td + n_r, e_r \Downarrow h', k_0, v, (\delta_r, m_r, s_r)
\end{array}$$

where $h_r = h|_{dom\ h - \{p\}}$. Again, the induction hypothesis and Theorem 1 may be applied in order to get $\Delta_r \succeq_{h_r, k_0, Reg} \delta_r$, $\mu_r \succeq m_r$ and $\sigma_r \succeq s_r$.

1. $\Delta \succeq \delta$. From the inference rules we have $\Gamma\ x = T@\rho$ and $h\ (E\ x) = (j, C_r\ \overline{v_i}^{n_r})$. Hence the binding $[\rho \mapsto j]$ belongs to $build(h, E\ x, \Gamma\ x)$. Since $\rho \in dom\ \Delta$, we get $\rho \in dom\ Reg$ and hence $[\rho \mapsto j] \in Reg$.

$$\begin{array}{rcl}
\sum\limits_{Reg\ \rho'=j} (\Delta\ \rho'\ \overline{s_i}^n) & = & \sum\limits_{\substack{Reg\ \rho'=j \\ \rho' \neq \rho}} (\Delta\ \rho'\ \overline{s_i}^n) + \Delta\ \rho\ \overline{s_i}^n \\
& = & \sum\limits_{\substack{Reg\ \rho'=j \\ \rho' \neq \rho}} (\max\{\Delta_i\ \rho'\ \overline{s_i}^n | 1 \leq i \leq l\}) \\
& & \quad + \max\{\Delta_i\ \rho\ \overline{s_i}^n | 1 \leq i \leq l\} - 1 \\
& = & \sum\limits_{Reg\ \rho'=j} (\max\{\Delta_i\ \rho'\ \overline{s_i}^n | 1 \leq i \leq l\}) - 1 \\
& \geq & \sum\limits_{Reg\ \rho'=j} (\Delta_r\ \rho'\ \overline{s_i}^n) - 1 \\
& \geq & \delta_r\ j - 1 \\
& = & \delta\ j
\end{array}$$

With respect to the remaining regions $i \in \{0 \dots k_0\} - \{j\}$, we can proceed similarly as in the nondestructive **case**.

2. $\mu \succeq m$.

$$\begin{array}{rcl}\mu\ \overline{s_i}^n & = & \max\{0, \sqcup_{i=1}^{l} \mu_i\ \overline{s_i}^n\} \\ & = & \max\{0, \max\{\mu_i\ \overline{s_i}^n - 1 \mid 1 \le i \le l\}\} \\ & \geq & \max\{0, \mu_r\ \overline{s_i}^n - 1\} \\ & \geq & \max\{0, m_r - 1\} \\ & = & m\end{array}$$

3. $\sigma \succeq s$. The proof given for the nondestructive **case** may be applied here.

$\square$

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 3. *Let f be a context function. Given Σ_1, Σ_2, Γ, 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. $\square$

6 Space Inference Algorithms

Given a recursive function f with $n+m$ arguments, the algorithms for inferring Δ_f and σ_f do not depend on each other, while the algorithm for inferring μ_f needs a correct value for Δ_f. We will assume that μ_f, σ_f, and the cost functions in Δ_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 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. 7(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 Counting the number of recursive calls

An important precondition for the correctness of the algorithms described in the following sections is the fact that the nr_f, nb_f and len_f are upper bounds of the actual number of recursive and base calls, and the maximum number of nested calls. In order to take these figures into account we add extra annotations to the big-step operational semantics of Figure 1. We will have judgments of the form:

$$E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s), (n_t, n_b, l)_f$$

where n_t is the total number of calls to f occurring in the evaluation of e (including the current call, since we assume that f is the context function) from which n_b calls correspond to base cases. The number of

$$E \vdash h,k,td,c \Downarrow h,k,c,(1,1,1)_f \; [Lit]$$

$$E[x \mapsto v] \vdash h,k,td,x \Downarrow h,k,v,(1,1,1)_f \; [Var]$$

$$\frac{j \leq k \quad (h',p') = copy(h,p,j) \quad m = size(h,p)}{E[x \mapsto p, r \mapsto j] \vdash h,k,td,x@r \Downarrow h',k,p',(1,1,1)_f} \; [Var_2]$$

$$\frac{fresh(q)}{E[x \mapsto p] \vdash h \uplus [p \mapsto w],k,td,x! \Downarrow h \uplus [q \mapsto w],k,q,(1,1,1)_f} \; [Var_3]$$

$$\frac{g \neq f \quad (g\ \overline{x_i}^n\ @\ \overline{r_j}^t = e) \in \Sigma \quad [\overline{x_i \mapsto E(a_i)}^n, \overline{r_j \mapsto E(r'_j)}^t, self \mapsto k+1] \vdash h,k+1,n+l,e \Downarrow h',k+1,v,(n_t,n_b,l)_f}{E \vdash h,k,td,g\ \overline{a_i}^n\ @\ \overline{r'_j}^t \Downarrow h'|_k,k,v,(n_t,n_b,l)_f} \; [App-NonRec]$$

$$\frac{(f\ \overline{x_i}^n\ @\ \overline{r_j}^t = e) \in \Sigma \quad [\overline{x_i \mapsto E(a_i)}^n, \overline{r_j \mapsto E(r'_j)}^t, self \mapsto k+1] \vdash h,k+1,n+l,e \Downarrow h',k+1,v,(n_t,n_b,l)_f}{E \vdash h,k,td,f\ \overline{a_i}^n\ @\ \overline{r'_j}^t \Downarrow h'|_k,k,v,(n_t+1,n_b,l+1)_f} \; [App-Rec]$$

$$\frac{\begin{array}{c} E \vdash h,k,0,e_1 \Downarrow h',k,v_1,(n_{t1},n_{b1},l_1)_f \\ E \cup [x_1 \mapsto v_1] \vdash h',k,td+1,e_2 \Downarrow h'',k,v,(n_{t2},n_{b2},l_2)_f \end{array}}{E \vdash h,k,td,\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \Downarrow h'',k,v,(n_{t1}+n_{t2}-1, n_{b1} \oplus_{n_{t1},n_{t2}} n_{b2}, \max\{l_1,l_2\})_f} \; [Let_1]$$

$$\frac{j \leq k \quad fresh(p) \quad E \cup [x_1 \mapsto p] \vdash h \uplus [p \mapsto (j, C\ \overline{v_i}^n)],k,td+1,e_2 \Downarrow h',k,v,(n_t,n_b,l)_f}{E[\overline{a_i \mapsto v_i}^n, r \mapsto j] \vdash h,k,td,\textbf{let}\ x_1 = C\ \overline{a_i}^n @r\ \textbf{in}\ e_2 \Downarrow h',k,v,(n_t,n_b,l)_f} \; [Let_2]$$

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

$$\frac{C = C_r \quad E \cup [\overline{x_{r_i} \mapsto v_i}^{n_r}] \vdash h,k,td+n_r,e_r \Downarrow h',k,v,(n_t,n_b,l)_f}{E[x \mapsto p] \vdash h \uplus [p \mapsto (j, C\ \overline{v_i}^n)],k,td,\textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \Downarrow h',k,v,(n_t,n_b,l)_f} \; [Case!]$$

Figure 4: Big-step operational semantics enriched with number of calls

recursive childs in the call tree can be obtained by substracting n_b from n_t. The maximum number of nested calls is reflected in l.

The resulting rules are shown in Figure 4. The (δ, m, s) annotations are left out for simplicity. All of them require no explanation, except the one corresponding to **let** expressions. In this case we sum the number of total calls from each subexpression and subtract 1 (otherwise we would count the actual call twice). With regard to the resulting n_b, if both subexpressions contain recursive calls we just add the corresponding n_b's, otherwise we only consider the number of base calls of the subexpression not having recursive calls. This is specified by means of the $\oplus$ operator, defined as follows:

$$x \oplus_{n_{t1},n_{t2}} y = \begin{cases} x & \text{if } n_{t2} = 1 \\ y & \text{if } n_{t1} = 1 \\ x + y & \text{e.o.c} \end{cases}$$

By simple inspection of the rules one can prove that $n_t \geq n_b$ and hence the expression $n_{b1} \oplus_{n_{t1},n_{t2}} n_{b2}$ in $[Let_1]$ is well-defined. The following Lemma shows an important property of these annotations.

Lemma 4. *Let e be an expression such that the following judgment holds for some E, h, k, td_i, h',v, δ, m, s, n_t, n_b and l:*

$$E \vdash h,k,td,e \Downarrow h',k,v,(\delta,m,s),(n_t,n_b,l)_f \tag{7}$$

Let us assume that there are p direct recursive calls to f in the derivation of (7). That is, for each $i \in \{1 \dots p\}$ there exist some E_i, h_i, h'_i, v_i, δ_i, m_i, s_i, $n_{t,i}$, n_{b_i} and l_i such that:

$$E_i \vdash h_i,k+1,td_i,e_f \Downarrow h'_i,k+1,v_i,(\delta_i,m_i,s_i),(n_{t,i},n_{b,i},l_i)_f$$

belongs to (7). Therefore it holds that:

$$n_t = 1 + \sum_{i=1}^{p} n_{t,i} \qquad n_b = \sum_{i=1}^{p} n_{b,i}$$

6.2 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. 5 describes the functions *splitExp* and *splitAlt* which, given a *Safe*

$$
\begin{array}{l}
\mathit{splitExp}_f\ [\![e]\!] = (e, \#) \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{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) \\
\quad \mathbf{where} \quad (e_{1b}, e_{1r}) = \mathit{splitExp}_f\ [\![e_1]\!] \\
\quad\quad (e_{2b}, e_{2r}) = \mathit{splitExp}_f\ [\![e_2]\!] \\
\quad\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} \\
\quad\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) \\
\quad \mathbf{where} \quad (\overline{alt_{ib}}^n, \overline{alt_{ir}}^n) = \mathit{unzip}\ (\mathit{map}\ \mathit{splitAlt}_f\ \overline{alt_i}^n) \\
\quad\quad e_b = \begin{cases} \# & \text{if } alt_{ib} = \# \to \# \text{ for all } i \in \{1 \dots n\} \\ \mathbf{case}(!)\ x\ \mathbf{of}\ \overline{alt_{ib}}^n & \text{otherwise} \end{cases} \\
\quad\quad e_r = \begin{cases} \# & \text{if } alt_{ir} = \# \to \# \text{ for all } i \in \{1 \dots 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) \\
\quad \mathbf{where} \quad (e_b, e_r) = \mathit{splitExp}_f\ e \\
\quad\quad alt_b = \begin{cases} \# \to \# & \text{if } e_b = \# \\ C\ \overline{x_j}^n \to e_b & \text{otherwise} \end{cases} \\
\quad\quad alt_r = \begin{cases} \# \to \# & \text{if } e_r = \# \\ C\ \overline{x_j}^n \to e_r & \text{otherwise} \end{cases}
\end{array}
$$

Figure 5: Function splitting a *Core-Safe* expression into its base and recursive cases

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. 6 we define such function, called $\mathit{splitBA}_f$. In Fig. 7 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 $[\overline{(e^i_{bef}, e^i_{aft})}^n] = \mathit{splitBA}_f[\![e_r]\!]$.

Lemma 5. *Let* $(e_b, e_r) = \mathit{splitExp}_f\ e$*. Then,* $e_b \neq \#$ *and* $E \vdash h, k, td, e_b \Downarrow h', k, v, (\delta, m, s)$ *if and only if* $E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$ *such that there is no call to* f *in this derivation.*

Proof. Both implications can be proved by induction on the depth of the $\Downarrow$-derivation. We distinguish cases according to the structure of e for $(\Leftarrow)$ and e_b for $(\Rightarrow)$.

- **Cases** c, x, $x!$, $x@r$ and $C\ \overline{a_i}^n@r$

  Both implications hold trivially by hypothesis, by applying the same operational semantics rule since $e = e_b$ in all these cases.

- **Case** $g\ \overline{a_i}^n\ @\ \overline{r_j}^m$

  $(\Leftarrow)$ The absence of calls to f in the whole $\Downarrow$-derivation forces g to be distinct from f and in this case the implication holds trivially by hypothesis, since $e = e_b$.

  $(\Rightarrow)$ As $e_b \neq \#$, by definition of *splitExp* again $g \neq f$ and $e_b = e$, so the implication holds by hypothesis and because there is not mutual recursion in the language.

- **Case let**

  $(\Leftarrow)$ Let $e = \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2$. We get:

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

Figure 6: Function splitting a *Core-Safe* expression into its parts executing before and after the last recursive call

```
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
  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 #
```

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

```
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

Figure 7: Splitting a *Core-Safe* definition

$$
\begin{array}{ll}
(A) & E \vdash h, k, 0, e_1 \Downarrow h_1, k, v_1, (\delta_1, m_1, s_1)\\
(B) & E \cup [x_1 \mapsto v_1] \vdash h_1, k, td+1, e_2 \Downarrow h, k, v, (\delta_2, m_2, s_2)
\end{array}
$$

with $\delta = \delta_1 + \delta_2$, $m = \max\{m_1, |\delta_1| + m_2\}$ and $s = \max\{2 + s_1, 1 + s_2\}$. We know that in the derivations of both (A) and (B) there are no calls to f. Let $(e_{1b}, e_{1r}) = \mathit{splitExp}\ e_1$ and $(e_{2b}, e_{2r}) = \mathit{splitExp}\ e_2$. By induction hypothesis $e_{1b} \neq \#$, $e_{2b} \neq \#$, and

$$
\begin{array}{ll}
(A') & E \vdash h, k, 0, e_{1b} \Downarrow h_1, k, v_1, (\delta_1, m_1, s_1)\\
(B') & E \cup [x_1 \mapsto v_1] \vdash h_1, k, td+1, e_{2b} \Downarrow h, k, v, (\delta_2, m_2, s_2)
\end{array}
$$

Since both e_{1b} and e_{2b} are nonempty we get $e_b = \mathbf{let}\ x_1 = e_{1b}\ \mathbf{in}\ e_{2b} \neq \#$, and from the judgements (A') and (B') we can derive $E \vdash h, k, td, e_b \Downarrow h', k, v, (\delta, m, s)$.

(⇒) Let $e_b = \mathbf{let}\ x_1 = e_{1b}\ \mathbf{in}\ e_{2b}$. By definition of *splitExp*, $e = \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2$ where $(e_{1b}, \_) = \mathit{splitExp}\ e_1$ and $(e_{2b}, \_) = \mathit{splitExp}\ e_2$, and $e_{1b}, e_{2b} \neq \#$. Similarly to the proof of (⇐), this implication holds by applying induction hypothesis.

- **Case case(!)**

($\Leftarrow$) Let $e = \textbf{case}(!)\ x\ \textbf{of}\ \overline{alt_i}^n$, where $alt_i = C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i$. Assume $E(x) = p$ and $h(p) = (j, C_r\ \overline{v_j}^{n_r})$ for some $r \in \{1 \ldots n\}$. By the rules $[Case]$ and $[Case!]$ we get:

$$E \cup [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h_r, k, td + n_r, e_r \Downarrow h', k, v, (\delta_r, m_r, s_r)$$

where the relationships between h, δ, m, s and h_r, δ_r, m_r, s_r are given by the corresponding rule ($[Case]$ or $[Case!]$). Let $(e_{rb}, e_{rr}) = \mathit{splitExp}\ e_r$. Since in the derivation above for e_r there is no call to f, we can apply the induction hypothesis in order to ensure that $e_{rb} \neq \#$ and that:

$$E \cup [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h_r, k, td + n_r, e_{rb} \Downarrow h', k, v, (\delta_r, m_r, s_r)$$

Moreover, and since $e_{rb} \neq \#$ we get $e_b = \textbf{case}(!)\ x\ \textbf{of}\ \overline{alt_{ib}}^n \neq \#$ and we can derive $E \vdash h, k, td, e_b \Downarrow h', k, v, (\delta, m, s)$ by applying the same rule ($[Case]$ or $[Case!]$).

($\Rightarrow$) Let $e_b = \textbf{case}(!)\ x\ \textbf{of}\ \overline{alt_{ib}}^n$, where $alt_{ib} = C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_{ib}$. By definition of *splitExp*, $e = \textbf{case}(!)\ x\ \textbf{of}\ \overline{alt_i}^n$ such that $(alt_{ib}, \_) = \mathit{splitAlt}\ alt_i$ for each $i \in \{1..n\}$ and there exists at least one $s \in \{1..n\}$ such that $alt_{sb} \neq \#$.

By rule $[Case]$ or $[Case!]$, there exists $r \in \{1..n\}$ such that:

$$E \cup [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h_r, k, td + n_r, e_{rb} \Downarrow h', k, v, (\delta_r, m_r, s_r)$$

There is no operational rule for an empty expression, which implies that e_{rb} must be non-empty. By applying induction hypothesis on alternative r we get the desired implication, in a similar way to ($\Leftarrow$).

□

As we have introduced a new *Core-Safe* expression $\sqcup_i e_i$, we must give its big-step operational semantics. The following non-deterministic rule does this:

$$\frac{\exists j\ .\ E \vdash h, k, td, e_j \Downarrow h', k, v, (\delta, m, s)}{E \vdash h, k, td, \sqcup_i e_i \Downarrow h', k, v, (\delta, m, s)}\ [Lub]$$

Lemma 6. *Let $(e_b, e_r) = \mathit{splitExp}_f\ e$. Then, $e_r \neq \#$ and $E \vdash h, k, td, e_r \Downarrow h', k, v, (\delta, m, s)$ if and only if $E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$ such that there is at least one direct call to f in this derivation.*

Proof. Both implications can be proved by induction on the depth of the $\Downarrow$-derivation. We distinguish cases according to the structure of e for ($\Leftarrow$) and e_r for ($\Rightarrow$). For the proof of ($\Rightarrow$), we use the fact that the structure of e_r is the same as the structure of e with the exception of the $\sqcup$ case. But in this case we know that it always correspond to a **let** expression.

Cases c, x, $x!$, $x@r$, $C\ \overline{a_i}^n@r$ and $g\ \overline{a_i}^n\ @\ \overline{r_j}^m$ with $g \neq f$

These cases are trivial in both directions as the corresponding hypotheses are false.

Case $f\ \overline{a_i}^n\ @\ \overline{r_j}^m$

Both implications hold trivially by hypothesis, since $e = e_r$.

Case let

($\Leftarrow$) Let $e = \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2$. By the operational semantics, we get:

$$\begin{array}{ll} (A) & E \vdash h, k, 0, e_1 \Downarrow h_1, k, v_1, (\delta_1, m_1, s_1) \\ (B) & E \cup [x_1 \mapsto v_1] \vdash h_1, k, td + 1, e_2 \Downarrow h, k, v, (\delta_2, m_2, s_2) \end{array}$$

with $\delta = \delta_1 + \delta_2$, $m = \max\{m_1, |\delta_1| + m_2\}$ and $s = \max\{2 + s_1, 1 + s_2\}$. Let $(e_{1b}, e_{1r}) = \mathit{splitExp}\ e_1$ and $(e_{2b}, e_{2r}) = \mathit{splitExp}\ e_2$. We know that in the derivations of either (A), or (B), or both, there are direct calls to f. Let us distinguish these three cases:

1. There are calls in (A). By the induction hypothesis we get $e_{1r} \neq \#$ and:

$$(A') \qquad E \vdash h, k, 0, e_{1r} \Downarrow h_1, k, v_1, (\delta_1, m_1, s_1)$$

   As e_{1r} is non-empty, *splitExp* e gives either $e_r = \textbf{let } x_1 = e_{1r} \textbf{ in } e_2$ or:

$$e_r = \bigsqcup \{\textbf{let } x_1 = e_{1b} \textbf{ in } e_{2r},\ \textbf{let } x_1 = e_{1r} \textbf{ in } e_2\}$$

   In both cases we get $e_r \neq \#$ and $E \vdash h, k, td, e_r \Downarrow h', k, v, (\delta, m, s)$.
2. There are calls in (B) but not in (A). By the induction hypothesis $e_{2r} \neq \#$. The reasoning is symmetrical to the previous case.

($\Rightarrow$) Let $e_r = \textbf{let } x_1 = e_{1r} \textbf{ in } e_{2r}$. As $e_r \neq \#$, we have to distinguish two cases.

$e_{1r} = \#$, $e_{2r} \neq \#$ In this case $e_{1r} = e_1$ and $(\_, e_{2r}) =$ *splitExp* e_2. By hypothesis on e_1 and induction hypothesis on e_{2r} we prove this implication in a similar way to ($\Leftarrow$).

$e_{1r} \neq \#$, $e_{2r} = \#$ In this case $e_{2r} = e_2$ and $(\_, e_{1r}) =$ *splitExp* e_1. . The reasoning is symmetrical to the previous case.

Case case(!)

($\Leftarrow$) Let $e = \textbf{case}(!)\ x \textbf{ of } \overline{alt_i}^n$, where $alt_i = C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i$.
We assume $E(x) = p$ and $h(p) = (j, C_l\ \overline{v_j}^{n_r})$ for some $l \in \{1 \dots n\}$. By the rules [*Case*] and [*Case*!] we get:

$$E \cup [\overline{x_{lj} \mapsto v_j}^{n_l}] \vdash h_l, k, td + n_l, e_l \Downarrow h', k, v, (\delta_l, m_l, s_l)$$

where the relationships between h, δ, m, s and h_l, δ_l, m_l, s_l are given by the corresponding rule ([*Case*] or [*Case*!]). Let $(e_{lb}, e_{lr}) =$ *splitExp* e_l. Since in the derivation above for e_l there are calls to f, we can apply the induction hypothesis on e_l and get $e_{lr} \neq \#$ and:

$$E \cup [\overline{x_{lj} \mapsto v_j}^{n_l}] \vdash h_l, k, td + n_l, e_{lr} \Downarrow h', k, v, (\delta_l, m_l, s_l)$$

Moreover, and since $e_{lr} \neq \#$, by the definition of *splitExp*, we get $e_r = \textbf{case}(!)\ x \textbf{ of } \overline{alt_{ir}}^n$ and we can derive $E \vdash h, k, td, e_r \Downarrow h', k, v, (\delta, m, s)$ by applying the same rule ([*Case*] or [*Case*!]).

($\Rightarrow$) Let $e_r = \textbf{case}(!)\ x \textbf{ of } \overline{alt_{ir}}^n$, where $alt_{ir} = C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_{ir}$. By definition of *splitExp*, there exists $e = \textbf{case}(!)\ x \textbf{ of } \overline{alt_i}^n$ such that $(\_, alt_{ir}) =$ *splitAlt* alt_i for each $i \in \{1..n\}$ and there exists at least one $s \in \{1..n\}$ such that $alt_{sr} \neq \#$.
By rule [*Case*] or [*Case*!], there exists $l \in \{1..n\}$ such that:

$$E \cup [\overline{x_{lj} \mapsto v_j}^{n_l}] \vdash h_l, k, td + n_l, e_{lr} \Downarrow h', k, v, (\delta_l, m_l, s_l)$$

There is no operational rule for an empty expression, which implies that e_{lr} must be non-empty. By applying induction hypothesis on alternative r we get the desired implication, in a similar way to ($\Leftarrow$).

Case $e_r = \bigsqcup \left\{ \begin{array}{l} \textbf{let } x_1 = e_{1b} \textbf{ in } e_{2r} \\ \textbf{let } x_1 = e_{1r} \textbf{ in } e_2 \end{array} \right\}$

This case has no sense for ($\Leftarrow$). In this case $e = \textbf{let } x_1 = e_1 \textbf{ in } e_2$ where $(e_{1b}, e_{1r}) =$ *splitExp*$_f$ $[\![e_1]\!]$, $(e_{2b}, e_{2r}) =$ *splitExp*$_f$ $[\![e_2]\!]$ and both e_{1r} and e_{2r} are non-empty. By rule [*Lub*]

$$(1)\ E \vdash h, k, td, \textbf{let } x_1 = e_{1b} \textbf{ in } e_{2r} \Downarrow h', k, v, (\delta, m, s)$$

or

$$(2)\ E \vdash h, k, td, \textbf{let } x_1 = e_{1r} \textbf{ in } e_2 \Downarrow h', k, v, (\delta, m, s)$$

Consider first the case when (1) holds. Then

$$\begin{array}{ll} (A1) & E \vdash h, k, 0, e_{1b} \Downarrow h_1, k, v_1, (\delta_1, m_1, s_1) \\ (B1) & E \cup [x_1 \mapsto v_1] \vdash h_1, k, td + 1, e_{2r} \Downarrow h, k, v, (\delta_2, m_2, s_2) \end{array}$$

with $\delta = \delta_1 + \delta_2$, $m = \max\{m_1, |\delta_1| + m_2\}$ and $s = \max\{2 + s_1, 1 + s_2\}$. As there is no rule for an empty expression, e_{1b} must be non-empty, so by Lemma 5:

$$(A1') \qquad E \vdash h, k, 0, e_1 \Downarrow h_1, k, v_1, (\delta_1, m_1, s_1)$$

As e_{2r} is non-empty, by induction hypothesis

$$(B1') \qquad E \cup [x_1 \mapsto v_1] \vdash h_1, k, td + 1, e_2 \Downarrow h, k, v, (\delta_2, m_2, s_2)$$

and there is a call to f in this derivation. So we can derive:

$$E \vdash h, k, td, \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 \Downarrow h', k, v, (\delta, m, s)$$

and there is a call to f in this derivation.
If (2) holds, the reasoning is similar. The difference is that we reason by induction on $e_{1r} \neq \#$ and by hypothesis on e_2. In this case we do not need Lemma 5.

□

6.3 Algorithm for computing Δ_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, $\Delta_f \stackrel{\text{def}}{=} \Delta_r\ |_{\rho \neq \rho_{self}}\ \times nr_f + \Delta_b\ |_{\rho \neq \rho_{self}}\ \times nb_f$.

If we apply the abstract interpretation rules for the base cases of our `splitAt` example in Fig. 7(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. 7(d) we get $\Delta_r = [\rho \mapsto \lambda n\ x.1 \mid \rho \in \{\rho_1, \rho_2\}]$. The resulting $\Delta_{\mathtt{splitAt}}$ is shown in Fig. 10.

Lemma 7. *If nr_f, nb_f, and all the size functions belong to $\mathbb{F}$, then all functions in Δ_f belong to $\mathbb{F}$.*

Proof. This is a consequence of $\mathbb{F}$ being closed by the operations $\{+, \sqcup, *\}$. Notice that it is critical that the final cost charged by Δ_f to any particular region be non-negative, i.e. destruction may be allowed only if it is compensated by allocation. □

Lemma 8. *Δ_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 Δ_r, Δ_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 Δ_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 $\Delta_2 = \Delta\ |_{\rho \neq \rho_{self}}$.

Lemma 9. *For all n, $\mathbb{I}^n_\Delta(\Delta_f)$ is a correct abstract heap for f.*

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 Δ_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. We prove now that $\mathbb{I}_\Delta$ is reductive, i.e. $\mathbb{I}_\Delta(\Delta_f) \sqsubseteq \Delta_f$. Let us assume that there are n recursive calls to f in e_r and that $\overline{a_{ji}}$ are the arguments of the recursive call j. We also assume that region ρ_{self} is ignored in all the interpretations below:

$$
\begin{array}{ll}
& \pi_1(\mathbb{I}_\Delta(\Delta_f(\overline{x_i})), \_, \_) \\
= & \pi_1\,(\llbracket e_r \rrbracket\ \Sigma[f \mapsto \Delta_f]\ \Gamma\ (n+m)) \qquad \text{-- by definition of } \mathbb{I}_\Delta \\
= & \sum_{j=1}^{n} \Delta_f(\overline{a_{ji}}) + \Delta_r(\overline{x_i}) \qquad \text{-- rules for interpreting } \Delta \text{ are additive} \\
= & \sum_{j=1}^{n} (\Delta_r(\overline{a_{ji}}) \times nr(\overline{a_{ji}}) + \Delta_b(\overline{a_{ji}}) \times nb(\overline{a_{ji}})) + \Delta_r(\overline{x_i}) \qquad \text{-- by definition of } \Delta_f \\
\sqsubseteq & (\sqcup_{j=1}^{n} \Delta_r(\overline{a_{ji}}))(\sum_{j=1}^{n} nr(\overline{a_{ji}})) \\
& +(\sqcup_{j=1}^{n} \Delta_b(\overline{a_{ji}}))(\sum_{j=1}^{n} nb(\overline{a_{ji}})) + \Delta_r(\overline{x_i}) \qquad \text{-- mathematics} \\
\sqsubseteq & \Delta_r(\overline{x_i})(\sum_{j=1}^{n} nr(\overline{a_{ji}})) + \Delta_b(\overline{x_i})(\sum_{j=1}^{n} nb(\overline{a_{ji}})) + \Delta_r(\overline{x_i}) \qquad \text{-- } a_{ji} \sqsubseteq x_i \text{ and } \Delta_r, \Delta_b \text{ monotonic} \\
\sqsubseteq & \Delta_r(\overline{x_i})(nr(\overline{x_i}) - 1) + \Delta_b(\overline{x_i}) nb(\overline{x_i}) + \Delta_r(\overline{x_i}) \qquad \text{-- } \sum_{j=1}^{n} nr(\overline{a_{ji}}) \sqsubseteq nr(\overline{x_i}) - 1 \text{ and} \\
& \qquad \text{-- } \sum_{j=1}^{n} nb(\overline{a_{ji}}) \sqsubseteq nb(\overline{x_i}) \\
= & \Delta_f
\end{array}
$$

Notice the assumption on well-behaviour of functions nr and nb. □

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

6.4 Algorithm for computing μ_f

We separately infer the part μ_{self} of μ_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 $(\_, \mu_b, \_) = \llbracket e_b \rrbracket\ \Sigma\ \Gamma\ (n+m)$, i.e. the heap needs of the non-recursive part of $f's$ body.
3. Let $([\rho_{self} \mapsto \mu_{self}], \_, \_) = \llbracket e_{bef} \rrbracket\ \Sigma\ \Gamma\ (n+m)$, i.e. the charges to ρ_{self} made by the part of $f's$ body before (and including) the last recursive call.
4. Let $(\_, \mu_{bef}, \_) = (\llbracket e_{bef} \rrbracket\ \Sigma\ \Gamma\ (n+m))|_{\rho \neq \rho_{self}}$, i.e. the heap needs of $f's$ body before the last recursive call, without considering the *self* region.
5. Let $(\_, \mu_{aft}, \_) = \llbracket e_{aft} \rrbracket\ \Sigma\ \Gamma\ (n+m)$, 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. 7, 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 10. *If the functions in Δ_f, len_f, and the size functions belong to $\mathbb{F}$, then μ_f belongs to $\mathbb{F}$.*

Proof. This is a consequence of $\mathbb{F}$ being closed by the operations $\{+, \sqcup, *\}$ and $len_f \sqsupseteq 1$. □

Lemma 11. *μ_f is a safe upper bound for f's heap needs.*

Proof. (Proof sketch)

1. $|\,\Delta_f\,|$ is a safe upper bound of the live memory during the evaluation of f, observed at any point of f's body and disregarding ρ_{self}, because it is the live memory at f's end.
2. $\mu_{self} \times (len_f - 1)$ is an upper bound of the live memory at ρ_{self} when executing the last but one call of the longest f's call chain.
3. $\sqcup\{\mu_{bef}, \mu_b, \mu_{aft}\}$ is an upper bound of the peak memory needed by all regions but ρ_{self} before calling f for the last time, and of the peak memory needed in all regions by the last call to f, and of the peak memory needed in all regions when returning from the last call and executing the 'after' portion of the previous call to f.

In turn, all this is a consequence of the correctness of the abstract interpretation rules, and of Δ_f, len_f, and the size functions being upper bounds of their respective runtime figures. □

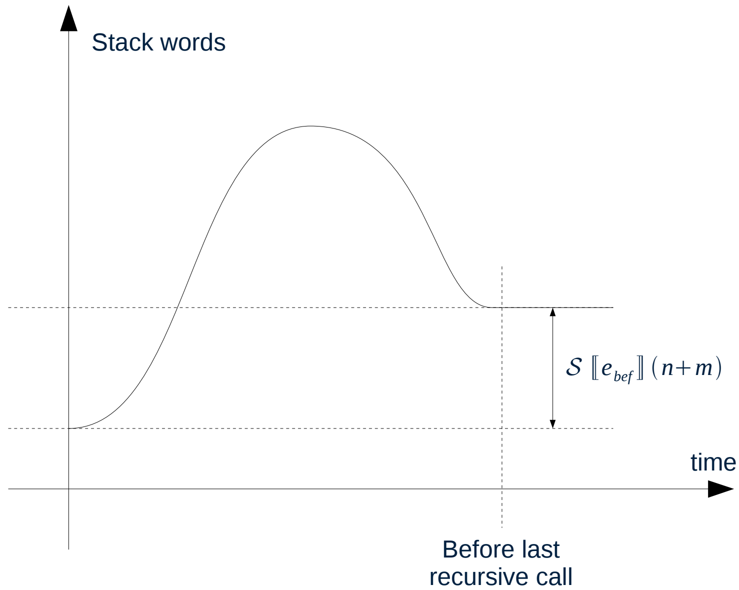


Figure 8: Intuitive meaning of the $\mathcal{S}$ function

As in the case of Δ_f, we can define an interpretation $\mathbb{I}_\mu$ taking any upper bound μ_1 as input, and producing a better one $\mu_2 = \mathbb{I}_\mu(\mu_1)$ as output.

Lemma 12. *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 μ_f. We prove now that $\mathbb{I}_\mu$ is reductive, i.e. $\mathbb{I}_\mu(\mu_f) \sqsubseteq \mu_f$. For simplicity, let us assume that there is only one recursive call to f in e_r and that $\mu', \Delta', \ldots$ denote the corresponding functions $\mu, \Delta, \ldots$ applied to the arguments $\overline{a_i}$ of the recursive call.

$$
\begin{array}{lll}
 & \pi_2(\_, \mathbb{I}_\mu(\mu_f), \_) & \\
= & \pi_2\,([\![e_r]\!]\ \Sigma[f \mapsto \mu_f]\ \Gamma\ (n+m)) & \text{-- by definition of } \mathbb{I}_\mu \\
= & |\,\Delta'_f\,| + \mu'_{self} \times (len'_f - 1) + \sqcup\{\mu'_{bef}, \mu'_b, \mu'_{aft}\} + |\,\Delta_r\,| + \mu_{self} & \text{-- rules for interpreting } \mu \text{ are additive} \\
\sqsubseteq & |\,\Delta_f\,| + \mu'_{self} \times (len'_f - 1) + \sqcup\{\mu'_{bef}, \mu'_b, \mu'_{aft}\} + \mu_{self} & \text{-- } nr'_f \sqsubseteq nr_f - 1 \text{ implies } \Delta'_f \sqsubseteq \Delta_f - \Delta_r \\
\sqsubseteq & |\,\Delta_f\,| + \mu_{self} \times (len_f - 1) + \sqcup\{\mu'_{bef}, \mu'_b, \mu'_{aft}\} & \text{-- } len'_f \sqsubseteq len_f - 1 \\
\sqsubseteq & |\,\Delta_f\,| + \mu_{self} \times (len_f - 1) + \sqcup\{\mu_{bef}, \mu_b, \mu_{aft}\} & \text{-- } a_i \sqsubseteq x_i \text{ and } \mu_{bef}, \mu_b, \mu_{aft} \text{ monotonic} \\
= & \mu_f &
\end{array}
$$

Notice the assumption on well-behaviour of function *len*. □

6.5 Algorithm for computing σ_f

The algorithm for inferring μ_f traverses f's body from left to right because the abstract interpretation rules for μ need the charges to the previous heaps. For inferring σ_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 (Fig. 8):

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

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

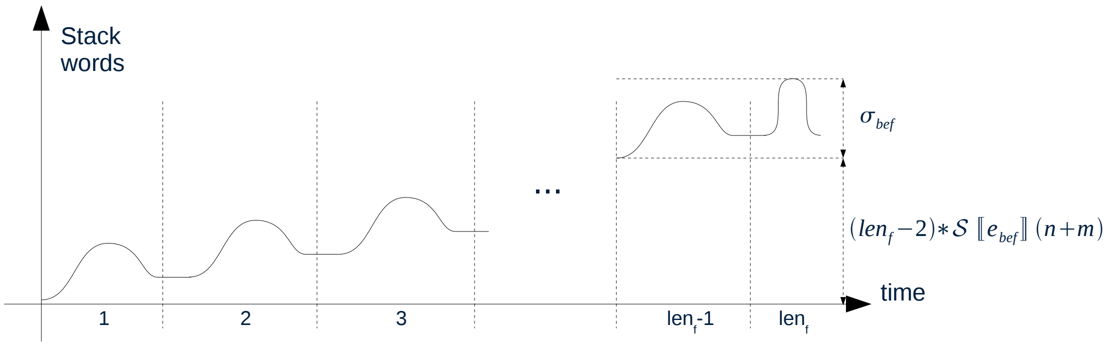
In our example, if we denote by $e_{\mathit{bef}}^{\mathtt{splitAt}}$ the definition of Fig. 7(b) we get $\mathcal{S}\ [\![e_{\mathit{bef}}^{\mathtt{splitAt}}]\!]\ (2+3) = 9$ and, by applying the abstract interpretation rules to the definitions in Fig. 7(c),(b) and (e) we obtain $\sigma_b = \lambda n\ x.4$, $\sigma_{\mathit{bef}} = \lambda n\ x.13$ and $\sigma_{\mathit{aft}} = \lambda n\ x.9$. Hence $\sigma_f = 9\min\{n-1, x-2\} + 13 = 9\min\{n, x-1\} + 4$.

Lemma 13. *If len_f, and all the size functions belong to $\mathbb{F}$, then σ_f belongs to $\mathbb{F}$.*

Proof. The result of $\mathcal{S}\ [\![e_f]\!]\ \mathit{td}$ is nonnegative when $\mathit{td} = n + m$. Moreover, the results of σ_{bef}, σ_{aft} and σ_b are monotonic functions. □

Lemma 14. *σ_f is a safe upper bound for f's stack needs.*

Proof. (Sketch) 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.



The result of $\mathcal{S}\ [\![e_{\mathit{bef}}]\!]\ (n+m) * (\mathit{len}_f - 2)$ is an upper bound to the stack length before the last recursive case, since we are taking into account the maximum number of nested recursive calls and words pushed between calls. The term $\sqcup\{\sigma_{\mathit{bef}}, \sigma_b, \sigma_{\mathit{aft}}\}$ correctly approximates the stack cost of the last but one recursive call. □

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

7 Case Studies

In Fig. 9 we show a *Full-Safe* version of the mergesort algorithm (the code for `splitAt` was presented in Fig. 7) with the types inferred by the compiler. Region ρ_1 is used inside `msort` for the internal call `splitAt n' xs @ r1 r1 self`, while region ρ_2 receives the charges made by `merge`. Notice that some charges to `msort`'s *self* region are made by `splitAt`. In Fig. 10 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 Δ and σ, and another three for μ by using the last Δ. The upper bounds for Δ and

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

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
```

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

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

Figure 9: *Full-Safe* mergesort program

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

Figure 10: Cost results for the mergesort program

μ 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 Δ. Also, μ and σ decrease in the second iteration but not in the third. This confirms the predictions of lemmas 9 and 12.

We have tried some more examples and the results inferred for μ and σ after a maximum of three iterations are shown in Fig. 11, 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. 12). 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. 12), 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 [15] 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 [16, 14] 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

Function	Heap needs μ	Stack needs σ
$partition(p, x)$	$3x - 1$	$9x - 5$
$append(x, y)$	$x - 1$	$\max(8, 7x - 6)$
$quicksort(x)$	$3x^2 - 20x + 76$	$\max(40, 20x - 27)$
$insertD(e, x)$	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

Figure 11: 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!
```

Figure 12: Two summation functions and a destructive tree insertion function

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^{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] 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.

[15] 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.

[16] 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.