# Space Consumption Analysis by Abstract Interpretation
## Reductivity Properties ☆

Manuel Montenegro, Ricardo Peña, Clara Segura

*Departamento de Sistemas Informáticos y Computación*
*Universidad Complutense de Madrid*

**Abstract**

In a previous paper we presented an abstract interpretation-based static analysis for inferring heap and stack memory consumption in a functional language. The language, called *Safe*, is eager and first-order, and its memory management system is based on heap regions instead of the more conventional approach of having a garbage collector.

In this paper we concentrate on an important property of our analysis, namely that the inferred bounds are *reductive* under certain reasonable conditions. This means that by iterating the analysis using as input the prior inferred bound, we can get tighter and tighter bounds, all of them correct. In some cases, even the exact bound is obtained.

The paper includes several examples and case studies illustrating in detail the reductivity property of the inferred bounds.

*Keywords:* resource analysis, abstract interpretation, functional languages, regions.

## 1. Introduction

In a previous paper [13], we presented an abstract interpretation-based static analysis for inferring heap and stack memory consumption in a functional language. It describes the automatic analysis of memory bounds for a

☆Work supported by the projects TIN2008-06622-C03-01/TIN (STAMP), TIN2009-14599-C03-01 (DESAFIOS10), S2009/TIC-1465 (PROMETIDOS) and the MEC FPU grant AP2006-02154.

*Email addresses:* `montenegro@fdi.ucm.es` (Manuel Montenegro), `ricardo@sip.ucm.es` (Ricardo Peña), `csegura@sip.ucm.es` (Clara Segura)

first-order functional language called *Safe*, which has been developed in the last few years as a research platform for analysing and formally certifying properties of programs with regard to memory usage. Memory consumption is especially relevant for instance when programming embedded devices. It is necessary to ensure that programs will not stop due to lack of memory. It is also useful to know in advance how much memory will be needed in order to reduce hardware and energy costs.

The first results on memory consumption analysis were targeted towards the functional programming paradigm. The developed techniques were subsequently adapted to mainstream languages, such as Java or C++. Hughes and Pareto introduce in [8] a first-order functional language with a type and effect system guaranteeing termination and execution in bounded space. This system is a combination of Tofte and Talpin's approach to regions and of sized types [9, 19]. The first fully automatic way to infer closed-form memory bounds is due to Hofmann and Jost [7]. Their analysis, based on a type system with resource annotations, can infer linear heap memory bounds on first-order functional programs with explicit deallocation. This approach is extended in [11, 10] to higher-order programs. More recently, Hoffmann and Hofmann have extended their initial work [7] to polynomial memory bounds [6, 5], and Simões et al [20] have further extended it to lazy evaluation.

The COSTA system [2] follows a different approach, since it generates recurrence equations and provides its own recurrence relation solver, PUBS [1], which can handle multivariate, non-deterministic recurrence relations. COSTA is an abstract interpretation-based analyser which works at the level of Java bytecode, and supports several notions of cost, such as the number of executed bytecode instructions, heap consumption, and number of calls to a particular method.

In order to better compute memory bounds, we have decided *Safe* to have a heap structured as a region stack in which regions are allocated and deallocated in constant time. Given this memory model, we used in [13] abstract interpretation-based techniques for inferring non-linear, monotonic, closed-form expressions bounding the heap and stack memory costs of a program. *Safe*'s type system [12] also supports *polymorphic recursion on regions*, meaning in essence that the recursive internal calls may use different regions than the external call. The language provides also a *destructive pattern matching* feature which allows the programmer to explicitly dispose data structures, or parts of them. Both features result in programs with less memory consumption. Due to technical difficulties, we have excluded them in this work,

but we provide more comments on this in Section 5.

Since the memory needs of a program usually depend on its input, the bounds we obtain in our analysis are multivariate functions on the sizes of the inputs. Even if we restrict ourselves to a first-order functional language like *Safe*, the inference of safe memory bounds is a very complex task, which involves considering several preliminary results, such as size analysis, and call-tree size analysis. Each one of these analyses is by itself a subject of extensive research. In the analysis developed in [13] the size and call-tree information is given externally.

The full development of the analyis can be found in [16], which is an extended and improved version of [13]. We can summarize the original contributions of [13] as follows: (1) it infers space bounds for a functional language with lexically scoped regions; (2) it uses abstract interpretation directly on the infinite domain of multivariate monotonic functions; and (3) the bounds go beyond multivariate polynomials. The additional contributions of this paper are the following:

- Under certain mild conditions on the externally-given call-tree information, our bounds have the nice property of *reductivity*. This means that if we use a correct bound as an input of the abstract interpretation, we get a new bound which is not only correct but also at least as tight.
- We have formally proved the correctness of all the results.
- We have implemented all the algorithms presented here in our *Safe* compiler.

The proofs of the theorems, including the statement and proof of some auxiliary lemmas, are included in [17].

*Plan of the paper*

After this introduction, in Section 2 we summarise *Safe*'s features, the abstract intepretation rules, and the space inference algorithms already presented in [13, 16]. Then, Section 3 is devoted to the reductivity property. We specify and prove under which conditions this property holds. Detailed examples of reductivity are included here. Section 4 presents some medium-sized case studies and the results obtained for many other functions. Section 5 discusses how it is possible to extend these results when polymorphic recursion and explicit destruction are considered. Finally, Section 6 concludes.

## 2. Preliminaries

### *2.1. The Safe language*

*Safe* is a first-order eager functional language with a Haskell-like syntax, but with a different approach to memory management. Instead of using a garbage collector, *Safe*'s heap memory model is based on a combination of *regions* and *explicit destruction*. The latter aspect is controlled by the programmer and it is not considered in this paper. However, in Section 5 we sketch the inference of memory consumption in presence of this feature.

A region is a part of the memory, disjoint from other regions, in which data structures are built. In this work, we define a data structure (DS in the following) as the set of cells that stem from a given one (the root cell) and have the same type. A cell is just a piece of memory containing a constructor and the arguments to which it is applied. A naive implementation of cells will assign a fixed number of bytes to all of them, but more clever implementations are possible [15]. We pose the restriction that a data structure must be contained within a single region. For instance, assume a list of lists of integer values, of type $[[Int]]$. The cons-nil spine of the outer list is a DS which may reside in a region different from those of the inner lists, each of which is a separate DS. The integer values of the inner lists do not belong to any region by themselves; they are contained within cells.

Regions are created and destroyed in a stack-like fashion. There is a correspondence between the function call stack and the region stack. The region associated with a function call to $f$ is called its *working region*. An empty region is allocated when a function call to $f$ starts, and it is deallocated when this call finishes. In the meanwhile new cells can be created in this region at anytime, either by the owner function $f$ or by some other function called from it. All the DSs contained within a region are disposed of in constant time when the region is deallocated. Regions are not handled directly by a *Safe* programmer, but inferred by the compiler [14], which decorates function and constructor applications with additional arguments: region variables. These contain, at runtime, natural numbers denoting region positions in the stack, being number zero the bottommost region, i.e. the first one being created.

Function definitions are also decorated with additional region parameters. These are separated from the rest of normal parameters by a @ and contain the region(s) in which the result of the function will be built.

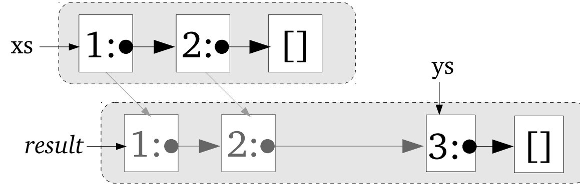



Figure 1: Function *append*: the result is built on the region of the second parameter

**Example 1.** Consider the following function for concatenating two lists.

$$\begin{array}{lllcl} append & [\,] & ys \ @\ r & = & ys \\ append & (x : xs) & ys \ @\ r & = & (x \ : \ append\ xs\ ys\ @\ r)\ @\ r \end{array}$$

There is a new region parameter $r$, which is used to build the resulting list, and is passed to the subsequent recursive calls. The memory behaviour of *append* is illustrated in Figure 1. □

Notice that, in the previous example, the result list shares the list passed as second argument. If the programmer wants to avoid this sharing, *Safe* provides a built-in facility for copying data structures. The expression $ys\ @\ r$ returns a copy of the recursive spine of $ys$, which will be located in the region given by $r$. If we substitute, in the example above, $ys\ @\ r$ for $ys$ in the right-hand side of the first equation, the result of *append* would not point to the list $ys$, but to a copy of it. From now on, let us denote by *appendC* this variant of the *append* function.

There exists a special region variable (*self*) in the scope of every function definition which contains the identifier of the working region of that function.

**Example 2.** Consider the following implementation of the *tree sort* algorithm:

$$treeSort\ xs\ @\ r = inorder\ (mkTree\ xs\ @\ self)\ @\ r$$

The *mkTree* function builds a binary search tree from a given list, and *inorder* builds a list by doing an inorder traversal of the elements of a given binary search tree. Both functions receive a parameter specifying where to build the result. The *treeSort* function builds the auxiliary binary search tree in its working region (*self*), and it generates the result from this tree in the output region given by $r$. When *treeSort* finishes, its working region disappears

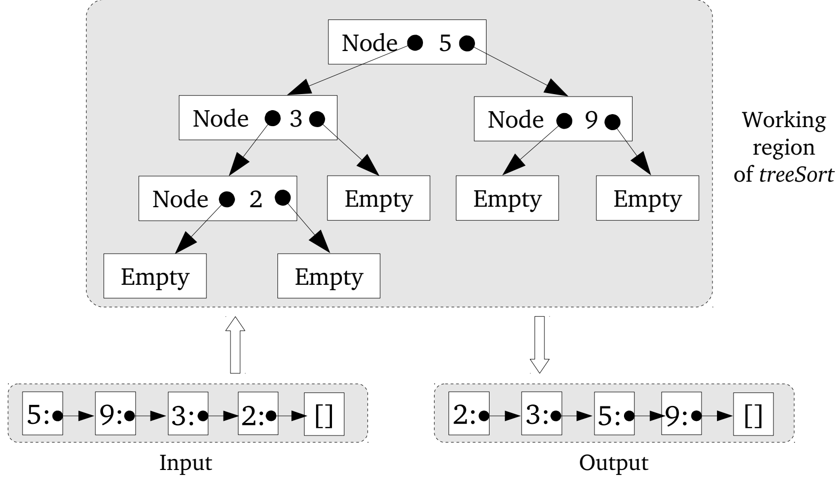



Figure 2: Regions used by the *treeSort* algorithm

from the heap, and so does the auxiliary tree. This behaviour is graphically illustrated in Figure 2. The full code of *tree sort* is shown in Figure 17, in Section 4. □

Our space consumption analysis is applied to programs written in a desugared variant of *Full-Safe*, whose name is *Core-Safe*. The syntax of the latter is shown in Figure 3. There, we use the abbreviation $\overline{item}$ to denote the sequence $item_1, \ldots, item_n$, and the notation $|\overline{item}|$ to denote its length. A program is a sequence of **data** declarations, followed by a sequence of function definitions, and a main expression. Notice that the syntax distinguishes between basic expressions and compound expressions, which respectively belong to **BExp** and **Exp**. The transformation from *Full-Safe* to *Core-Safe* is standard and it shall not be described here. As an example, the *append* function of Example 1 is translated as follows:

$$\begin{array}{rcl} append\ xs\ ys\ @\ r & = & \mathbf{case}\ xs\ \mathbf{of} \\ & & \quad [\,] \rightarrow ys \\ & & \quad (x : xx) \rightarrow \mathbf{let}\ x_1 = append\ xx\ ys\ @\ r\ \mathbf{in}\ (x : x_1)@r \end{array} \tag{1}$$

We have defined a big-step operational semantics for *Core-Safe* which also accounts for the heap and stack consumption when executing an expression.

$$
\begin{array}{rlll}
\textbf{Prog} \ni & prog & \rightarrow & \overline{data}; \overline{def}; e \\
\textbf{DecData} \ni & data & \rightarrow & \textbf{data}\ T\ \overline{\alpha}\ @\ \overline{\rho} = \overline{altData} \\
& altData & \rightarrow & C\ \overline{t}\ @\ \rho \\
\textbf{DecFun} \ni & def & \rightarrow & f\ \overline{x}\ @\ \overline{r} = e \\
& & & \qquad\qquad\qquad \{\text{Atoms}\} \\
& a & \rightarrow & \ c \qquad\qquad\qquad \{\text{literal constant}\} \\
& & & \mid x \qquad\qquad\quad \{\text{variable}\} \\
& & & \qquad\qquad\qquad \{\text{Basic Expressions}\} \\
\textbf{BExp} \ni & be & \rightarrow & \ a \qquad\qquad\qquad \{\text{atom}\} \\
& & & \mid x\ @\ r \qquad\qquad \{\text{copy}\} \\
& & & \mid a \oplus a \qquad\qquad \{\text{basic operator application}\} \\
& & & \mid C\ \overline{a}\ @\ r \qquad\quad \{\text{constructor application}\} \\
& & & \mid f\ \overline{a}\ @\ \overline{r} \qquad\quad\ \{\text{function application}\} \\
& & & \qquad\qquad\qquad \{\text{Expressions}\} \\
\textbf{Exp} \ni & e & \rightarrow & \ be \qquad\qquad\quad \{\text{basic}\} \\
& & & \mid \textbf{let}\ x = e\ \textbf{in}\ e \quad \{\text{nonrecursive, monomorphic}\} \\
& & & \mid \textbf{case}\ x\ \textbf{of}\ \overline{alt} \quad \{\text{pattern matching}\} \\
& alt & \rightarrow & \ C\ \overline{x} \rightarrow e
\end{array}
$$

Figure 3: *Core-Safe* language definition.

This resource-aware semantics is beyond the scope of this paper, and we refer to [15, 16] for more details.

### *2.2. Safe's type system*

*Safe* provides an standard Hindley-Milner type system extended with *region type variables*, which are the types of region variables. A region type variable (RTV in the following) behaves as a polymorphic type variable in Haskell, but only region variables are allowed to have a RTV as its type. The compiler annotates algebraic data types with RTVs specifying the type of the region which was used for building the corresponding DSs. For instance, if $r$ has the RTV $\rho$ as its type, $[\,]@r$ has type $[\alpha]@\rho$. If the types of two variables are decorated with the same RTV, their corresponding DSs will live in the same region at runtime. For instance, the compiler infers the following type for the *append* function, and its variant *appendC*:

$$
\begin{array}{rcl}
append & :: & [\alpha]@\rho_1 \rightarrow [\alpha]@\rho_2 \rightarrow \rho_2 \rightarrow [\alpha]@\rho_2 \\
appendC & :: & [\alpha]@\rho_1 \rightarrow [\alpha]@\rho_2 \rightarrow \rho_3 \rightarrow [\alpha]@\rho_3
\end{array}
$$

In the first case the result must be located at the same region as the second parameter, whereas with *appendC* the result may live in a different region. In both cases, the result is built in the region specified by the third parameter (i.e. the region variable). *Safe*'s type system allows polymorphic recursion on regions. This is not dealt with in our space analysis [13, 16], but in Section 5 we briefly describe how we could handle it.

The **data** declarations section follows a syntax similar to that of Haskell, with the exception of the RTVs generated by the compiler. For instance, binary search trees are decorated as follows:

$$\textbf{data}\ \mathit{BSTree}\ \alpha\ @\ \rho = \mathit{Empty}\ @\ \rho \mid \mathit{Node}\ (\mathit{BSTree}\ \alpha\ @\ \rho)\ \alpha\ (\mathit{BSTree}\ \alpha\ @\ \rho)\ @\ \rho$$

As an example, we show below the type signatures of the functions occurring in the tree sort example.

$$\begin{array}{rcl} \mathit{mkTree} & :: & [\mathit{Int}]@\rho_1 \to \rho_2 \to \mathit{BSTree}\ \mathit{Int}\ @\ \rho_2 \\ \mathit{inorder} & :: & \mathit{BSTree}\ \alpha\ @\ \rho_1 \to \rho_2 \to [\alpha]@\rho_2 \\ \mathit{treeSort} & :: & [\mathit{Int}]@\rho_1 \to \rho_2 \to [\mathit{Int}]@\rho_2 \end{array}$$

A DS can be spread among several regions, whose types will be reflected in its algebraic type. For instance, a datatype of key-value pairs will be decorated as **data** $\mathit{TBL}\ \alpha\ \beta\ @\ \rho_1\ \rho_2\ \rho_3 = \mathit{TBL}\ [(\alpha, \beta)@\rho_1]@\rho_2\ @\ \rho_3$.

*2.3. Space consumption analysis by abstract interpretation*

*2.3.1. Abstract domain*

The space analysis of [13, 16] computes, for a function definition $f\ \overline{x}\ @\ \overline{r} = e_f$, a triple $(\Delta, \mu, \sigma)$ of $n$-ary functions. The $\mu$ and $\sigma$ components represent the minimum amount of memory cells (resp. stack words) which must be available to execute $f$, as a function on the size of the input arguments. For example, the *append* function builds as many cells as its first argument has, except the `[]` cell of the latter.

The size of an integer is defined as its value, whereas the size of a DS is the number of cells of its recursive spine. For instance, the size of a list of $s$ elements is $s + 1$ (a cell with the $[\,]$ constructor, plus $s$ cells with the (:) constructor), and the size of a *BSTree* with $m$ elements is $2m + 1$ ($m$ cells with the *Node* constructor plus $m+1$ cells with the *Empty* constructor). The $\mu$ and $\sigma$ components are assumed to belong to the following domain:

$$\mathbb{F}_f \stackrel{\text{def}}{=} \{\xi : ((\mathbb{R}^+_\infty)^\perp)^n \to (\mathbb{R}^+_\infty)^\perp \mid \xi \text{ is monotonic and strict}\}$$

where $\mathbb{R}^+_\infty \stackrel{\text{def}}{=} \mathbb{R}^+ \cup \{+\infty\}$, and the notation $D^\perp$ abbreviates $D \cup \{\perp\}$. The $+\infty$ value denotes the absence of an upper bound (either because it does not exist, or because the algorithm is not able to infer it), and $\perp$ denotes an undefined input value result (for instance, the memory needs of the *head* function are undefined when applied to an empty list). The order relation $\sqsubseteq$ on this set is defined in a pointwise basis (i.e. $\xi \sqsubseteq \xi'$ iff $\xi\ \overline{x} \leq \xi'\ \overline{x}$ for every $\overline{x}$). It is easy to show that $(\mathbb{F}, \sqsubseteq)$ is a complete lattice. We shall use a guarded $\lambda$ notation to denote elements from this lattice. For instance $\lambda xs.[xs \geq 2 \rightarrow xs - 1]$ is the function that, given an $xs$[1] returns $xs - 1$ if $xs$ is greater or equal than two, and $\perp$ otherwise.

The $\Delta$ component of the triple $(\Delta, \mu, \sigma)$ is also a function on the size of the parameters, but it returns a *mapping* from the types of the region parameters $\overline{r}$ to numbers in $\mathbb{R}^+_\infty$. Every RTV is associated with the charges done to each region during the execution of $f$, that is, the difference between the number of cells in a region when the execution of $f$ finishes and the number of cells in that region when the execution of $f$ starts. The $\Delta$ components are assumed to belong to a domain $\mathbb{D}_f$ defined as follows:

$$\mathbb{D}_f = \{\Delta : ((\mathbb{R}^+_\infty)^\perp)^n \rightarrow \left(R_f \rightarrow \mathbb{R}^+_\infty\right)^\perp \mid \Delta \text{ is monotonic and strict}\}$$

where $R_f$ is the set of the region types of the $\overline{r}$ parameters. The elements of $\mathbb{D}_f$ are called *abstract heaps*. In the context of expressions we shall also consider the charges done to the working region (of type $\rho_{self}$). In this context, abstract heaps belong to a domain $\mathbb{D}^*_f$ defined as above, but substituting $R_f \cup \{\rho_{self}\}$ for $R_f$. We can extend the above-mentioned $\sqsubseteq$ relation, and the usual $+$ and $*$ operators to abstract heaps in a standard way, and it turns out that $(\mathbb{D}_f, \sqsubseteq)$ is a complete lattice. Notice that both $\mathbb{F}_f$ and $\mathbb{D}_f$ are parametric on the function definition $f$ being inferred. Since the latter is easily deduced from the context, we shall just write $\mathbb{D}$ and $\mathbb{F}$ in the following.

### *2.3.2. Abstract interpretation rules*

Once we have defined the elements of our abstract domain, our next step is to devise a way to infer a triple $(\Delta, \mu, \sigma)$. Firstly we address this problem in the context of expressions by using an abstract interpretation function that, given an expression $e$, yields a triple $(\Delta, \mu, \sigma)$, where $\Delta \in \mathbb{D}^*$, and $\mu, \sigma \in \mathbb{F}$.

[1] In what follows, we will use the same name both for the argument of a *Safe* function and its size. Then, if $xs$ represents a list, we will also name $xs$ to its size.

$$\frac{}{[\![a]\!]\ \Sigma\ \Gamma\ td = ([\,], 0, 1)}$$

$$\frac{}{[\![a_1 \oplus a_2]\!]\ \Sigma\ \Gamma\ td = ([\,], 0, 2)}$$

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

$$\frac{}{[\![C\ \overline{a}\ @\ r]\!]\ \Sigma\ \Gamma\ td = ([\Gamma(r) \mapsto 1], 1, 1)}$$

$$\frac{\begin{array}{c}\Sigma(g) = (\Delta_g, \mu_g, \sigma_g) \qquad \theta = \mathit{unify}(\Gamma, g, \overline{r}) \qquad y_i = |a_i|\ \overline{x} \\ G = (\Delta_g\ \overline{y} \neq \bot \wedge \mu_g\ \overline{y} \neq \bot \wedge \sigma_g\ \overline{y} \neq \bot) \qquad l = |\overline{x}| \qquad q = |\overline{r}| \\ \Delta = [G \rightarrow \theta \downarrow_{\overline{y}} \Delta_g] \qquad \mu = [G \rightarrow \mu_g\ \overline{y}] \qquad \sigma = [G \rightarrow \sigma_g\ \overline{y}]\end{array}}{[\![g\ \overline{a}\ @\ \overline{r}]\!]\ \Sigma\ \Gamma\ td = (\Delta, \mu, \sqcup\{l+q, l+q-td+\sigma\})}$$

Figure 4: Abstract interpretation for basic expressions.

The abstract interpretation of $e$ is parametric on a signature environment $\Sigma$ which maps each function name $g$ occurring in $e$ to a signature $(\Delta_g, \mu_g, \sigma_g)$ specifying the memory consumption of its corresponding definition. It is also parametric on a typing environment $\Gamma$ that gives the RTVs of the region variables in scope, and a statically determined number $td$, which accounts for the number of words in the stack above the topmost continuation (see [16] for details).

In Figure 4 we define the abstract interpretation of basic expressions as a set of rules. For the sake of brevity, we omit the $\lambda\overline{x}$ prefix in the cost functions and abstract heaps of the right-hand side. An explanation for these rules can be found in [16]. In this paper, we only explain function application.

Assume we want to infer a particular function application $g\ \overline{a}\ @\ \overline{r}$. Firstly, we retrieve the signature of $g$ from the signature environment $\Sigma$, so let $(\Delta_g, \mu_g, \sigma_g) = \Sigma(g)$. Each component is a function which depends on the sizes of the $l$ parameters given to $g$, so we have to pass the sizes of the actual arguments $|a_i|$, which, in turn, are functions of the parameters of the caller (that is why we have $|a_i|\ \overline{x}$). The guard $G$ discards those values $\overline{x}$ leading to sizes $|a_i|\ \overline{x}$ not belonging to the domain of $\Delta_g$, $\mu_g$ or $\sigma_g$. We have to find a mapping $\theta$ between the RTVs in the $g$'s most general type and the RTVs of the particular instance used in the call $g\ \overline{a}\ @\ \overline{r}$. That is what the function *unify* does.

For instance, if in $\Gamma$ $g$ has type $t_1 \rightarrow t_2 \rightarrow \rho_1 \rightarrow \rho_2 \rightarrow t_3$ and $r$ has type $\rho$, and $g$ is called as $g\ x\ y\ @\ r\ r$, then $\mathit{unify}(\Gamma, g, [r, r])$ will return $\theta = \{\rho_1 \mapsto \rho, \rho_2 \mapsto \rho\}$.

If there are several RTVs of $g$'s type being mapped to the same RTV $\rho$

$$
\begin{array}{lcl}
[\![\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2]\!]_\sigma\ td & = & \sqcup\!\!\!\cdot\{2 + [\![e_1]\!]_\sigma\ 0, 1 + [\![e_2]\!]_\sigma\ (td+1)\} \\
[\![\textbf{case}\ x\ \textbf{of}\ \overline{alt}]\!]_\sigma\ td & = & \bigsqcup_{r=1}^{n} [|x| \geq 1 + NumRecPos(C_r) \to n_r + [\![e_r]\!]_\sigma\ (td + n_r)] \\
& & \textbf{where}\ n = |\overline{alt}| \\
& & \qquad\quad alt_r = C_r\ \overline{x}_r \to e_r \\
& & \qquad\quad n_r = |\overline{x}_r|
\end{array}
$$

Figure 5: Stack consumption of compound expressions.

in the function application, then the charges done to the region of type $\rho$ are the sum of the charges made by $g$ to the RTVs $\rho'$ such that $\theta(\rho') = \rho$. The $\downarrow$ operator does this computation. It is defined as follows

$$\theta \downarrow_{\overline{y}} \Delta = \lambda\rho. \sum_{\rho' \in \theta^{-1}(\rho)} \Delta\ \overline{y}\ \rho'$$

where $\rho \in R_f \cup \{\rho_{self}\}$.

We use the notation $[\![e]\!]_\Delta$, $[\![e]\!]_\mu$, and $[\![e]\!]_\sigma$ to refer to the first, second, and third components of $[\![e]\!]$, and we omit the $\Sigma$, $\Gamma$, and $td$ parameters when they are clear from the context.

Our next step is to define the result of the abstract interpretation when applied to compound expressions (**let** and **case**). In the context of stack consumption, we define the $[\![\cdot]\!]_\sigma$ interpretation as in Figure 5. This definition is analogous to the resource-aware semantics defined in [16]. The function *NumRecPos* returns the number of recursive positions in the constructor $C_i$, and the $\sqcup\!\!\!\cdot$ operator denotes a strict variant of the $\sqcup$ operator.

In the context of heap consumption, we transform (i.e. we flatten) the compound expression **let** or **case** into a *set of sequences* of basic expressions. Each sequence represents a possible execution flow through the compound expression, and it may be preceded by a guard $G$ with conditions on the sizes of the input arguments, which must hold in order to execute the sequence. We use the notation $[G \to be_1, \ldots, be_n]$ to denote such sequences. In [16] we defined a function *seqs* which, given a compound expression $e$, returns its corresponding set of sequences.

**Example 3.** If $e_{append}$ denotes the *Core-Safe* expression of the *append* function given in (1), this expression can be flattened as follows:

$$seqs\ e_{append} = \{[xs \geq 1 \to ys], [xs \geq 2 \to append\ xx\ ys, (x : x_1)@r]\}$$

$$
\begin{array}{lcllcl}
[\![e]\!]_\triangle & \stackrel{\text{def}}{=} & [\![seqs\ e]\!]_\triangle & [\![S]\!]_\triangle & \stackrel{\text{def}}{=} & \bigsqcup_{seq\in S}[\![seq]\!]_\triangle \\
[\![e]\!]_\mu & \stackrel{\text{def}}{=} & [\![seqs\ e]\!]_\mu & [\![S]\!]_\mu & \stackrel{\text{def}}{=} & \bigsqcup_{seq\in S}[\![seq]\!]_\mu
\end{array}
$$

$$
\begin{array}{lcl}
[\![[G \to be_1, ..., be_n]]\!]_\triangle & \stackrel{\text{def}}{=} & [G \to [\![be_1]\!]_\triangle + \ldots + [\![be_n]\!]_\triangle] \\
[\![[G \to be_1, ..., be_n]]\!]_\mu & \stackrel{\text{def}}{=} & [G \to \sqcup\!\!\!\cdot\,\{ \ [\![be_1]\!]_\mu, \\
& & \qquad \|[\![be_1]\!]_\triangle\| + [\![be_2]\!]_\mu, \\
& & \qquad \ldots, \\
& & \qquad \sum_{j=1}^{n-1} \|[\![be_j]\!]_\triangle\| + [\![be_n]\!]_\mu\}]
\end{array}
$$

Figure 6: Abstract interpretation of compound expressions and sequences.

□

The $[\![\cdot]\!]_\triangle$ and $[\![\cdot]\!]_\mu$ interpretations of a sequence of basic expressions is defined in Figure 6. If $\Delta$ is an abstract heap, the notation $\|\Delta\|$ denotes the cost function $\lambda\overline{x}.\sum_{\rho\in R_f\cup\{\rho_{self}\}} \Delta\ \overline{x}\ \rho$, which belongs to $\mathbb{F}$. In case the transformation of an expression gives rise to several sequences, we apply the interpretation to each of them, and take the least upper bound of the results. The definition of $[\![\cdot]\!]_\triangle$ for a sequence of basic expressions is straightforward since the incremental heap consumption is additive. The definition of $[\![\cdot]\!]_\mu$, i.e. the peak heap consumption approximation, for a sequence of basic expressions is graphically justified in Figure 7. In this definition we use the $\sqcup\!\!\!\cdot$ operator instead of $\sqcup$, since we want to make the cost of the whole sequence undefined when so is the cost of a single element in that sequence. For instance, given the following definition,

$$head\ (x : xs) = xs$$

the memory needs of the *head* function are given by the expression $\mu_{head} = [xs \geq 2 \to 0]$, as the *head* function yields an undefined result when applied to an empty list, which has size one. Therefore, if we flatten the expression ($\mathbf{let}\ ys = [\,]\ @\ r\ \mathbf{in}\ head\ ys$) into the sequence $[[\,]\ @\ r, head\ ys]$, and then apply the $[\![\cdot]\!]_\mu$ -intepretation to the latter, we would obtain $\sqcup\!\!\!\cdot\,\{1, 1+\bot\} = \bot$, as the expression we started from yields an undefined value.

**Example 4.** Back to our *append* function, assume a signature in which $\Sigma(append) = (f_\triangle, f_\mu, 0)$, for some $f_\triangle \in \mathbb{D}$ and $f_\mu \in \mathbb{F}$ such that dom $f_\triangle =$ dom $f_\mu = \{(xs, ys) \mid xs \geq 1\}$. The interpretation yields the following results:

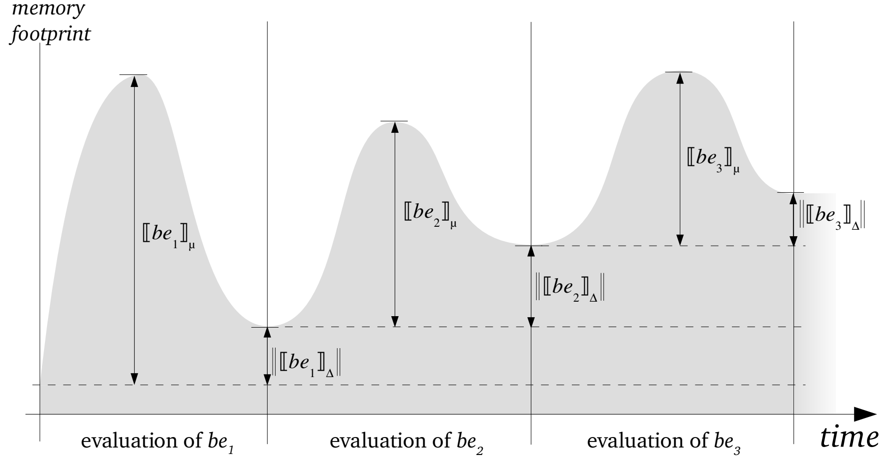



Figure 7: Peak heap consumption for a sequence of three basic expressions

$$
\begin{aligned}
[\![e_{append}]\!]_\Delta &= [xs \geq 2 \rightarrow [\rho_2 \mapsto 1 + f_\Delta\ (xs-1)\ ys\ \rho_2]] \sqcup \\
&\quad [xs \geq 1 \rightarrow [\rho_2 \mapsto 0]] \\
[\![e_{append}]\!]_\mu &= [xs \geq 2 \rightarrow \sqcup\{f_\mu\ (xs-1)\ ys, f_\Delta\ (xs-1)\ ys\ \rho_2 + 1\}] \sqcup \\
&\quad [xs \geq 1 \rightarrow 0]
\end{aligned}
$$

□

There exists a correctness result [16] that establishes the soundness of the bounds computed in this way, provided the signatures given in $\Sigma$ are correct bounds to their corresponding function definitions.

### *2.4. Inference of recursive definitions*

The abstract interpretation function defined in the previous section allows us to infer upper bounds to heap and stack memory consumption of nonrecursive function definitions. However, we cannot directly apply this abstract interpretation to recursive functions, as we would need to include in $\Sigma$ a correct bound to the recursive function, which is precisely what we are trying to obtain. If we find a triple $(\Delta_0, \mu_0, \sigma_0)$ of correct *initial bounds*, we may store it in the signature environment $\Sigma$ and apply the abstract interpretation in order to get more precise bounds. In this subsection we briefly

$$
\begin{array}{l}
\mathit{computeDelta}\ (f\ \overline{x}\ @\ \overline{r} = e_f)\ \Sigma\ \Gamma\ nb\ nr = (\lfloor\Delta_b\rfloor * nb + \lfloor\Delta_r\rfloor * nr) \sqcup \lfloor\Delta_b\rfloor \\
\quad \textbf{where}\ \begin{array}{lcl}
S & = & \mathit{seqs}\ e_f \\
(S_b, S_r) & = & \mathit{splitExp}_f\ S \\
\Delta_b & = & [\![S_b]\!]_\Delta\ \Sigma\ \Gamma \\
\Delta_r & = & [\![S_r]\!]_\Delta\ (\Sigma \uplus [f \mapsto ([\,], 0, 0)])\ \Gamma
\end{array}
\end{array}
$$

Figure 8: Algorithm for computing $\Delta_0$.

describe how to compute these initial bounds. A more detailed description of these algorithms can be found in [16]. In Section 3 we show that if we apply the abstract interpretation with these bounds in the signature environment $\Sigma$ we obtain a triple $(\Delta_1, \mu_1, \sigma_1)$ which is equal to or more precise than $(\Delta_0, \mu_0, \sigma_0)$.

In order to compute these initial bounds, we need some information regarding the number of base ($nb_f$) and recursive calls ($nr_f$) generated during a given call to $f$. We also need information on the maximum number of nested recursive calls ($len_f$), that is, the height of the call tree. Each of these components is a function on the size of the input, and hence belongs to $\mathbb{F}$. We assume these three functions to be given externally. In general these functions are not independent of each other. For instance, with linear recursion we get $nr_f = len_f - 1$ and $nb_f = 1$. However, we shall not assume a fixed relation between them. The computation of these three functions is closely related to the problem of termination and the computation of *ranking functions*. For instance, a sytem such as PUBS [3, 4] would be able to provide them in many examples.

An algorithm for computing our initial approximation $\Delta_0$ is given in Figure 8. We separate the set of sequences that results from the flattening of $e_f$ into a set $S_b$ of *base sequences* (those who do not contain a recursive call to $f$), and a set $S_r$ of *recursive sequences* (the remaining ones). That is what the $\mathit{splitExp}_f$ function does. Then we compute an upper bound $\Delta_b$ to the charges done in the execution of a base case and so we do with the recursive cases $\Delta_r$. If $\Delta$ is an abstract heap in $\mathbb{D}^*$, $\lfloor\Delta\rfloor$ denotes the abstract heap in $\mathbb{D}$ that results from disregarding the information of $\rho_{\mathit{self}}$ from $\Delta$. The result of $\lfloor\Delta_b\rfloor * nb$ accounts for the consumption done by all the base cases, and so does $\lfloor\Delta_r\rfloor * nr$ for the recursive cases. The addition of both yields the overall consumption of $f$ (Figure 9). The upper bound with $\lfloor\Delta_b\rfloor$ is justified in [16].

The computation of the initial $\mu_0$ is given in Figure 10. There we assume

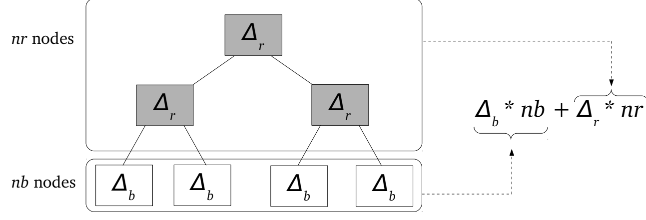



Figure 9: Approximation of the heap charges (Δ)

$$
\begin{array}{l}
\mathit{computeMu}\ (f\ \overline{x}\ @\ \overline{r} = e_f)\ \Sigma\ \Gamma\ \Delta\ \mathit{len} \\
\quad = (\Delta_{\mathit{self}} * (\mathit{len} - 1) + \|\Delta_{\mathit{bef}}\| + \sqcup\{\mu_{\mathit{bef}}, \mu_{\mathit{aft}}, \mu_b\}) \sqcup \mu_b \\
\qquad \textbf{where}\ \begin{array}[t]{lcl}
S & = & \mathit{seqs}\ e_f \\
(S_b, S_r) & = & \mathit{splitExp}_f\ S \\
(S_{\mathit{bef}}, S_{\mathit{aft}}) & = & \mathit{splitBA}_f\ S_r \\
\Delta^*_{\mathit{bef}} & = & [\![S_{\mathit{bef}}]\!]_\Delta\ (\Sigma \uplus [f \mapsto (\Delta, 0, 0)])\ \Gamma \\
\Delta_{\mathit{bef}} & = & \lfloor \Delta^*_{\mathit{bef}} \rfloor \\
\Delta_{\mathit{self}} & = & \Delta^*_{\mathit{bef}}\ \rho_{\mathit{self}} \\
\mu_{\mathit{bef}} & = & [\![S_{\mathit{bef}}]\!]_\mu\ (\Sigma \uplus [f \mapsto (\Delta, 0, 0)])\ \Gamma \\
\mu_{\mathit{aft}} & = & [\![S_{\mathit{aft}}]\!]_\mu\ (\Sigma \uplus [f \mapsto (\Delta, 0, 0)])\ \Gamma \\
\mu_b & = & [\![S_b]\!]_\mu\ (\Sigma \uplus [f \mapsto (\Delta, 0, 0)])\ \Gamma
\end{array}
\end{array}
$$

Figure 10: Algorithm for computing $\mu_0$.

the existence of a function *splitBA* that splits each recursive sequence *seq* of basic expressions in a set $S_r$ into two sequences: the first one containing the basic expressions being executed before (and including) the last recursive call, and the second one containing those being executed after the last recursive call. The function gathers all the 'before' parts in $S_{\mathit{bef}}$ and the 'after' parts in $S_{\mathit{aft}}$. The rationale behind the expression returned by *computeMu* is more involved than in the case of *computeDelta*, and we shall refer to [16] for more details.

In order to compute an initial approximation to the stack consumption $\sigma_0$, we have to approximate the number of words on the stack that can be inserted since the execution of a call starts, and until the next recursive call is about to be executed. This number is called the *stack level difference*, and it can be statically approximated by the $SD_f$ function. If we assume that this amount of words is inserted onto the stack along the execution of $\mathit{len}_f - 1$

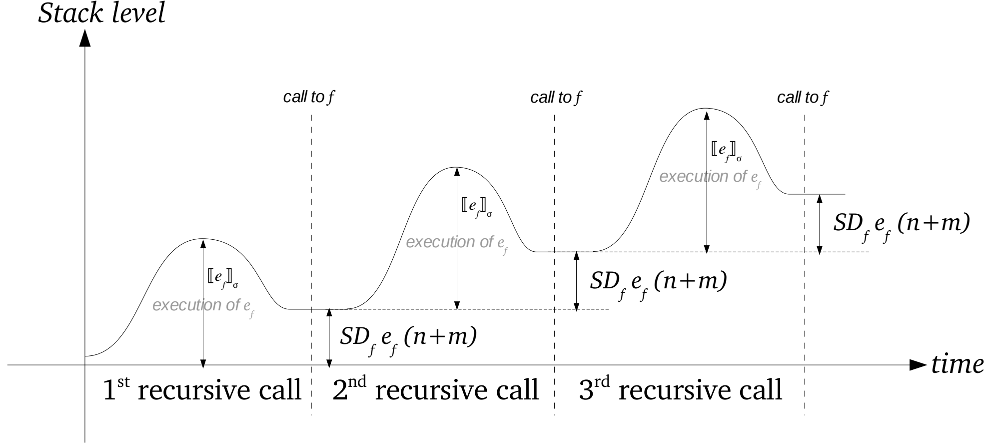



Figure 11: Function $SD_f$ and approximation of the stack consumption of a function $f$.

$$
\begin{array}{l}
computeSigma\ (f\ \overline{x}\ @\ \overline{r} = e_f)\ \Sigma\ td\ len \\
\quad = \sqcup\{0, SD_f\ e_f\ (n+m)\} * (len - 1) + \sigma \\
\quad\ \ \textbf{where}\ \ \begin{array}{lcl} \sigma & = & [\![e_f]\!]_\sigma\ (\Sigma \uplus [f \mapsto ([\,], 0, 0)])\ td \\ n & = & |\overline{x}| \\ m & = & |\overline{r}| \end{array}
\end{array}
$$

Figure 12: Computation of an initial approximation $\sigma_0$.

nested recursive calls, and then we add the maximum stack consumption of a single call, we get an initial approximation to the overall stack costs of a function definition. The algorithm is shown in Figure 12, and a graphical illustration of why this approximation is sound is depicted in Figure 11.

**Example 5.** Assume the following implementation of the *insertion sort* al-

gorithm:

$$
\begin{array}{l}
\mathit{insert} :: \mathit{Int} \to [\mathit{Int}]@\rho_1 \to \rho_2 \to [\mathit{Int}]@\rho_2 \\
\mathit{insert}\ y\ [\,] = [y] \\
\mathit{insert}\ y\ (x : xx)\ \mid y \leq x = y : x : xx \\
\qquad\qquad\qquad\quad \mid \textbf{otherwise} = x : \mathit{insert}\ y\ xx \\
\mathit{insSort} :: [\mathit{Int}]@\rho_1 \to \rho_2 \to [\mathit{Int}]@\rho_2 \\
\mathit{insSort}\ [\,] = [\,] \\
\mathit{insSort}\ (x : xx) = \mathit{insert}\ x\ (\mathit{insSort}\ xx)
\end{array}
$$

By applying the previously presented algorithms, we obtain the following initial approximation $(\Delta_{10}, \mu_{10}, \sigma_{10})$ for *insert*:

$$
\begin{array}{rcl}
\Delta_{10} & = & [zs \geq 2 \to [\rho_2 \mapsto zs + 1]] \sqcup [zs \geq 1 \to [\rho_2 \mapsto 2]] \\
\mu_{10} & = & [zs \geq 3 \to zs + 2] \sqcup [zs \geq 2 \to 4] \sqcup [zs \geq 1 \to 2] \\
\sigma_{10} & = & [zs \geq 2 \to 8zs] \sqcup [zs \geq 1 \to 8zs - 5]
\end{array}
$$

and the following initial approximation $(\Delta_{20}, \mu_{20}, \sigma_{20})$ for *insSort*:

$$
\begin{array}{rcl}
\Delta_{20} & = & [zs \geq 3 \to [\rho_2 \mapsto zs(zs - 1) + 1]] \sqcup [zs \geq 2 \to [\rho_2 \mapsto 2(zs - 1) + 1]] \sqcup \\
 & & [zs \geq 1 \to [\rho_2 \mapsto 1]] \\
\mu_{20} & = & [zs \geq 4 \to zs^2 - 2zs + 4] \sqcup [zs \geq 3 \to 2zs + 1] \sqcup [zs \geq 2 \to 3] \sqcup \\
 & & [zs \geq 1 \to 1] \\
\sigma_{20} & = & [zs \geq 3 \to 14zs - 13] \sqcup [zs \geq 2 \to \sqcup\{6zs, 14zs - 18\}] \sqcup \\
 & & [zs \geq 1 \to 6zs - 5]
\end{array}
$$

□

The *computeDelta*, *computeMu*, and *computeSigma* algorithms have been proven correct w.r.t. our resource-aware operational semantics, given that the abstract cost functions involved in their computations (i.e. $\Delta_b$, $\Delta_r$, etc) are parameter-decreasing [16]. A cost function is parameter-decreasing when it yields decreasing values from the root call to its recursive calls.

## 3. Reductivity properties

### *3.1. Preliminaries on fixed points in complete lattices*

Let $(L, \sqsubseteq)$ be a complete lattice and $f$ a monotonic function on $L$. Given an element $x \in L$, we say that: $x$ is a *fixed point* of $f$ iff $f(x) = x$; $f$ is *reductive* at $x$ iff $f(x) \sqsubseteq x$; and $f$ is *extensive* at $x$ iff $f(x) \sqsupseteq x$. We denote by $\mathit{Fix}(f)$ the set of fixed points of $f$. Similarly, we denote by $\mathit{Red}(f)$ (resp. $\mathit{Ext}(f)$) the set of points upon which $f$ is reductive (resp. extensive).

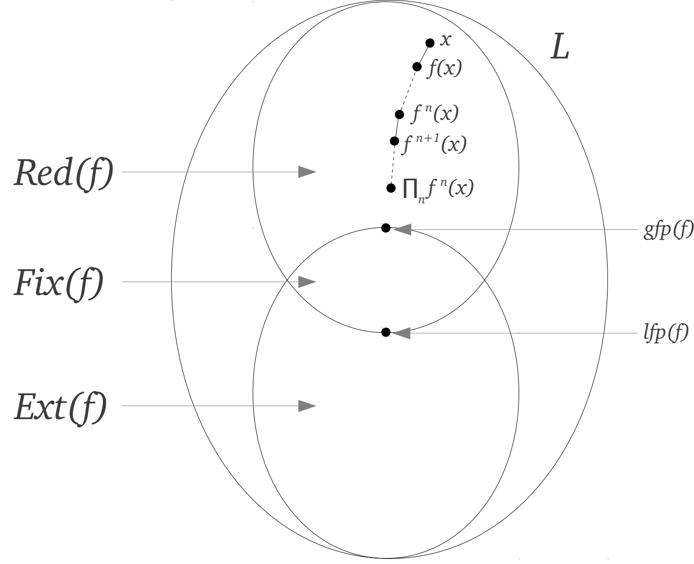



Figure 13: Representation of the points upon which a given function is reductive and extensive.

Given the fact that $L$ is a complete lattice, the least upper bound and the greatest lower bounds of $\mathit{Fix}(f)$ are both defined and respectively denoted by $\mathit{lfp}(f)$ and $\mathit{gfp}(f)$. Tarski's fixed point theorem [21] establishes the relation between fixed points and the reductivity/extensivity properties.

**Theorem 1 (Tarski 1955, taken from [18][Section A.4]).** *Let $(L, \sqsubseteq)$ a complete lattice and $f : L \to L$ a monotone function. Then $\mathit{lfp}(f) = \sqcap \mathit{Red}(f)$ and $\mathit{gfp}(f) = \sqcup \mathit{Ext}(f)$.*

Figure 13 depicts the layout of the reductive and extensive elements of $L$ w.r.t. a function $f$. If $x$ is an element belonging to $\mathit{Red}(f)$, then $f(x) \sqsubseteq x$. By monotonicity of $f$, we get $f(f(x)) \sqsubseteq f(x)$. By repeating this process we get the following chain,

$$x \sqsupseteq f(x) \sqsupseteq f^2(x) \sqsupseteq \ldots \sqsupseteq f^n(x) \sqsupseteq \ldots$$

whose elements, by Tarski's theorem, are located above the least fixed point.

*3.2. Iteration of our abstract interpretation*

Now let us apply these concepts to our particular abstract interpretation. Recall that $(\mathbb{F}, \sqsubseteq)$ and $(\mathbb{D}, \sqsubseteq)$ are complete lattices. Given the context function definition $f$, and some fixed $\Sigma$, $\Gamma$ and $td$, the iteration of the abstract

interpretation can be understood as a function on abstract heaps $\mathbb{D} \to \mathbb{D}$ (resp. on cost functions $\mathbb{F} \to \mathbb{F}$) which, given an input $\Delta$ (resp. $\mu$, $\sigma$), inserts it into the signature environment $\Sigma$, and applies the $[\![e_f]\!]_\Delta$ interpretation (resp. $[\![e_f]\!]_\mu$ and $[\![e_f]\!]_\sigma$) to the body of $f$.

**Definition 1.** Assume a function definition $f\ \overline{x}\ @\ \overline{r} = e_f \in \mathbf{FD}$, where $n = |\overline{x}|$ and $m = |\overline{r}|$, and some fixed $\Sigma$, $\Gamma$, $\Delta$, such that $f \notin \text{dom}\ \Sigma$. We define its *iteration operators* $\mathcal{D}_f : \mathbb{D} \to \mathbb{D}$, $\mathcal{M}_{\Delta,f} : \mathbb{F} \to \mathbb{F}$ and $\mathcal{S}_f : \mathbb{F} \to \mathbb{F}$ as follows:

$$
\begin{array}{rcl}
\mathcal{D}_f(\Delta) & = & \lfloor [\![e_f]\!]_\Delta\ (\Sigma \uplus [f \mapsto (\Delta, 0, 0)])\ \Gamma \rfloor \\
\mathcal{M}_{\Delta,f}(\mu) & = & [\![e_f]\!]_\mu\ (\Sigma \uplus [f \mapsto (\Delta, \mu, 0)])\ \Gamma \\
\mathcal{S}_f(\sigma) & = & [\![e_f]\!]_\sigma\ (\Sigma \uplus [f \mapsto ([\,]_f, 0, \sigma)])\ \Gamma\ (n+m)
\end{array}
$$

Notice that, strictly speaking, the $\mathcal{D}$, $\mathcal{M}$ and $\mathcal{S}$ operators are also parametric on the given $\Sigma$ and $\Gamma$, but we assume all these elements fixed. Now we prove their monotonocity.

**Proposition 1.** *The iteration operators $\mathcal{D}_f$, $\mathcal{M}_{\Delta,f}$ and $\mathcal{S}_f$ are monotonic on their input arguments.*

For each of these operators, we can define its set of fixed points, reductive elements and extensive elements in the same way as above.

**Example 6.** Consider the *insert* function of Example 5, and its iteration operator $\mathcal{D}_{insert}$, obtained by definition:

$$
\mathcal{D}_{insert}(\Delta) = [zs \geq 2 \to [\rho_2 \mapsto 1 + \Delta\ y\ (zs - 1)\ \rho_2]] \sqcup [zs \geq 1 \to [\rho_2 \mapsto 2]]
$$

Let us consider the abstract heaps $\Delta_1$, $\Delta_2$ and $\Delta_3$ defined as follows:

$$
\begin{array}{rcl}
\Delta_1 & \stackrel{\text{def}}{=} & [zs \geq 1 \to [\rho_2 \mapsto 0]] \\
\Delta_2 & \stackrel{\text{def}}{=} & [zs \geq 1 \to [\rho_2 \mapsto \lfloor zs \rfloor + 1]] \\
\Delta_3 & \stackrel{\text{def}}{=} & [zs \geq 1 \to [\rho_2 \mapsto 2zs]]
\end{array}
$$

If we apply the $\mathcal{D}_{insert}$ iterator to each of these, we obtain the following results:

$$
\begin{array}{rcl}
\mathcal{D}_{insert}(\Delta_1) & = & [zs \geq 2 \to [\rho_2 \mapsto 1 + 0]] \sqcup [zs \geq 1 \to [\rho_2 \mapsto 2]] \\
& = & [zs \geq 1 \to [\rho_2 \mapsto 2]]
\end{array}
$$

$$
\begin{aligned}
\mathcal{D}_{insert}(\Delta_2) &= [zs \geq 2 \rightarrow [\rho_2 \mapsto 1 + (\lfloor zs - 1 \rfloor + 1)]] \sqcup \\
&\quad\; [zs \geq 1 \rightarrow [\rho_2 \mapsto 2]] \\
&= [zs \geq 2 \rightarrow [\rho_2 \mapsto \lfloor zs \rfloor + 1]] \sqcup [zs \geq 1 \rightarrow [\rho_2 \mapsto 2]] \\
&= [zs \geq 1 \rightarrow [\rho_2 \mapsto \lfloor zs \rfloor + 1]]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{D}_{insert}(\Delta_3) &= [zs \geq 2 \rightarrow [\rho_2 \mapsto 1 + 2(zs - 1)]] \sqcup [zs \geq 1 \rightarrow [\rho_2 \mapsto 2]] \\
&= [zs \geq 2 \rightarrow [\rho_2 \mapsto 2zs - 1]] \sqcup [zs \geq 1 \rightarrow [\rho_2 \mapsto 2]]
\end{aligned}
$$

Therefore, $\Delta_1 \in \mathit{Ext}(\mathcal{D}_{insert})$, $\Delta_2 \in \mathit{Fix}(\mathcal{D}_{insert})$, and $\Delta_3 \in \mathit{Red}(\mathcal{D}_{insert})$.
□

*3.3. Reductivity of $\Delta_0$, $\mu_0$ and $\sigma_0$*

Now we will prove that:

1. The abstract heap $\Delta_0$ computed by *computeDelta* falls into the reductive area of the lattice $(\mathbb{D}, \sqsubseteq)$ with respect to the iteration of the abstract interpretation $\mathcal{D}_f$, provided some admissibility conditions on the externally given *nb* and *nr* functions hold.
2. The abstract costs $\mu_0$ and $\sigma_0$, respectively computed by *computeMu* and *computeSigma*, fall into the reductive area of the lattice $(\mathbb{F}, \sqsubseteq)$ with respect to the iteration of the respective abstract interpretations $\mathcal{M}_{\Delta,f}$ and $\mathcal{S}_f$ provided some admissibility conditions of the *len* function hold.

The admissibility conditions are first defined.

**Definition 2 (Admissible *nb*).** A function *nb* for computing the number of base calls is admissible with respect to a definition $f\ \overline{x}\ @\ \overline{r} = e_f \in \mathbf{FD}$ iff for every $\overline{x} \in \mathbb{R}^n$ the following conditions hold:

1. $nb\ \overline{x} \geq 1$
2. $\forall seq \in seqs\ e_f. \sum \{nb\ \overline{y} \mid f\ \overline{a}\ @\ \overline{r} \in seq \land y_i = |a_i|\ \overline{x}\} \leq nb\ \overline{x}$

**Definition 3 (Admissible *nr*).** A function *nr* for computing the number of recursive calls is admissible with respect to a definition $f\ \overline{x}\ @\ \overline{r} = e_f \in \mathbf{FD}$ iff for every $\overline{x} \in \mathbb{R}^n$ the following conditions hold:

1. $nr\ \overline{x} \geq 0$
2. $\forall seq \in seqs\ e_f. 1 + \sum \{nr\ \overline{y} \mid f\ \overline{a}\ @\ \overline{r} \in seq \land y_i = |a_i|\ \overline{x}\} \leq nr\ \overline{x}$

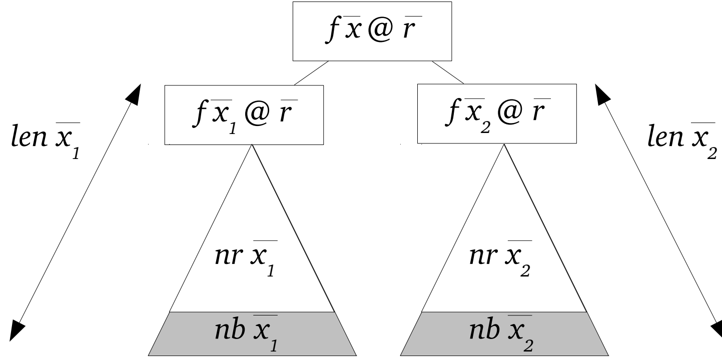



Figure 14: Activation tree of a call $f\ \overline{x}$ @ $\overline{r}$. The triangles represent the activation trees of each child call.

In order to get an intuitive idea on these conditions, let us consider the activation tree shown in Figure 14. The root of this tree is a call to $f$ with two recursive calls, each one with an approximation of its number of leafs $nb\ \overline{x_i}$ and internal nodes $nr\ \overline{x_i}$, with $i = 1, 2$. Hence, the whole tree has $nb\ \overline{x_1} + nb\ \overline{x_2}$ leafs or less. Admissibility on $nb$ holds if the approximation $nb\ \overline{x}$ given for the root node is greater than or equal to this number. Analogously, $nr$ is admissible if the approximation $nr\ \overline{x}$ of the root node is greater or equal than $1 + nr\ \overline{x_1} + nr\ \overline{x_2}$.

**Definition 4 (Admissible *len*).** A function *len* for computing the maximal length of call chains is admissible with respect to a definition $f\ \overline{x}$ @ $\overline{r}$ $= e_f \in$ **FD** iff for every $\overline{x} \in \mathbb{R}^n$ the following conditions hold:

1. $len\ \overline{x} \geq 1$
2. $\forall seq \in seqs\ e_f . 1 + \bigsqcup \{ len\ \overline{y} \mid f\ \overline{a}$ @ $\overline{r} \in seq \wedge y_i = |a_i|\ \overline{x} \} \leq len\ \overline{x}$
3. If $(S_b, S_r) = splitExp_f\ (seqs\ e_f)$, for every $seq \in S_r$, $len\ \overline{x} \geq 2$ whenever $guard(seq)\ \overline{x}$ holds.

The first admissibility condition is fairly reasonable. The second one is analogous to its counterparts in $nb$ and $nr$. Assume the situation given in Figure 14. An upper bound to the height of the whole tree is $1 + \sqcup\{len\ \overline{x_1}, len\ \overline{x_2}\}$. The second condition states that the approximated length $len\ \overline{x}$ must be greater than or equal to this bound. Finally, the third condition states that, in those values $\overline{x}$ that may lead to a recursive call, *len* must be greater than two (accounting for the caller and the callee).

**Theorem 2.** *Let us define* $\Delta_0 = \mathit{computeDelta}\ (f\ \overline{x}\ @\ \overline{r} = e_f)\ \Sigma\ \Gamma\ \mathit{nb}\ \mathit{nr}$*,* $\mu_0 = \mathit{computeMu}\ (f\ \overline{x}\ @\ \overline{r} = e_f)\ \Sigma\ \Gamma\ \Delta\ \mathit{len}$ *and* $\sigma_0 = \mathit{computeSigma}\ (f\ \overline{x}\ @\ \overline{r} = e_f)\ \Sigma\ (|\overline{x}| + |\overline{r}|)\ \mathit{len}$*.*

1. *If nr and nb are admissible, and the* $\Delta_b$ *and* $\Delta_r$ *occurring in the definition of computeDelta are parameter-decreasing, then:*

   $$\lfloor [\![ \mathit{seq} ]\!]_\Delta\ (\Sigma \uplus [f \mapsto (\Delta_0, 0, 0)])\ \Gamma \rfloor \sqsubseteq \Delta_0 \qquad \text{for every } \mathit{seq} \in \mathit{seqs}\ e_f$$

   *Therefore,* $\Delta_0 \in \mathit{Red}(\mathcal{D}_f)$
2. *If len is admissible, the* $\Delta_{\mathit{self}}$*,* $\mu_{\mathit{bef}}$*,* $\mu_{\mathit{aft}}$*,* $\mu_b$ *occurring in computeMu are parameter-decreasing, and* $\lfloor [\![ \mathit{seq} ]\!]_\Delta\ (\Sigma \uplus [f \mapsto (\Delta, 0, 0)])\ \Gamma \rfloor \sqsubseteq \Delta$ *for every* $\mathit{seq} \in \mathit{seqs}\ e_f$*, then:*

   $$[\![ \mathit{seq} ]\!]_\mu\ (\Sigma \uplus [f \mapsto (\Delta, \mu_0, 0)])\ \Gamma \sqsubseteq \mu_0 \qquad \text{for every } \mathit{seq} \in \mathit{seqs}\ e_f$$

   *Therefore,* $\mu_0 \in \mathit{Red}(\mathcal{M}_{\Delta,f})$*.*
3. *If len is admissible and the* $\sigma$ *occurring in the definition of computeSigma is parameter-decreasing, then* $\sigma_0 \in \mathit{Red}(\mathcal{S}_f)$*.*

Proof. We distinguish if *seq* is a base or a recursive sequence, and apply the abstract interpretation function.

This result allows us to iterate the abstract interpretation in order to reach more precise bounds by considering chains as the following,

$$\Delta_0 \sqsupseteq \mathcal{D}_f(\Delta_0) \sqsupseteq \mathcal{D}_f^2(\Delta_0) \sqsupseteq \cdots \sqsupseteq \mathcal{D}_f^n(\Delta_0) \sqsupseteq \cdots$$

which, if eventually stabilizes, it does in a fixed point. Since the abstract domain is infinite, the fixpoint is not necessarily reached in a finite number of iterations. Each new bound is at least as precise as the previous one but also usually more complex from the representation point of view, as will become apparent in the examples. Simplification algorithms could improve this although we have not implemented them yet.

Since the initial approximation to $\mu$ depends on the given input $\Delta$, it is advisable to spent some time iterating $\mathcal{D}_f$ in order to achieve better results. Notice that the $\Delta$ given as parameter to *computeMu* must be reductive for every sequence. This assumption holds, in particular, if $\Delta$ is either the initial approximation $\Delta_0$ computed by the algorithm of Section 2.4, or the result of applying the abstract interpretation $n > 0$ times to it, $\mathcal{D}_f^n(\Delta_0)$.

The following three examples show respectively how the iteration operators $\mathcal{D}_f$, $\mathcal{M}_{\Delta,f}$ and $\mathcal{S}_f$ work for functions *insert* and *insSort* of Example 5. In each case we show a couple of iterations and then a general term for the $i$-th iteration, which has been obtained by hand. The latter is used to obtain the limit of the sequence of functions, which in some cases is a fixpoint or coincides with the fixpoint in those sizes which are natural numbers.

**Example 7.** Assume the $\Delta_{10}$ of Example 5. By applying $\mathcal{D}_{insert}$ to this abstract heap we obtain:

$$\begin{array}{rcl}\mathcal{D}_{insert}(\Delta_{10}) & = & [zs \geq 3 \rightarrow [\rho_2 \mapsto zs+1]] \sqcup [zs \geq 2 \rightarrow [\rho_2 \mapsto 3]] \sqcup \\ & & [zs \geq 1 \rightarrow [\rho_2 \mapsto 2]]\end{array}$$

which is strictly smaller than $\Delta_{10}$ when $zs \in (2,3)$. Another iteration yields the following result:

$$\begin{array}{rclcll}\mathcal{D}^2_{insert}(\Delta_{10}) & = & [zs \geq 4 \rightarrow [\rho_2 \mapsto zs+1]] & \sqcup & [zs \geq 3 \rightarrow [\rho_2 \mapsto 4]] & \sqcup \\ & & [zs \geq 2 \rightarrow [\rho_2 \mapsto 3]] & \sqcup & [zs \geq 1 \rightarrow [\rho_2 \mapsto 2]] & \end{array}$$

In general, the $i$-th iteration results in the following abstract heap:

$$\mathcal{D}^i_{insert}(\Delta_{10}) = \begin{cases} \bot & zs < 1 \\ [\rho_2 \mapsto \lfloor zs \rfloor + 1] & 1 \leq zs < i+2 \\ [\rho_2 \mapsto zs + 1] & i+2 \leq zs \end{cases}$$

This sequence of functions converges to the fixpoint:

$$\Delta^{fix}_{insert} = [zs \geq 1 \rightarrow [\rho_2 \mapsto \lfloor zs \rfloor + 1]$$

In fact all the functions in the sequence coincide with $\Delta^{fix}_{insert}$ on the sizes which are natural numbers. In this case it is easy to prove that $\Delta^{fix}_{insert}$ is the only fixpoint of $\mathcal{D}_{insert}$, and that it represents the exact heap consumption of the worst case of *insert*.

Had we used a more precise function $nr_{insert} = [zs \geq 1 \rightarrow \lfloor zs \rfloor - 1]$ then we would have obtained directly a fixpoint:

$$\Delta'_{10} = [zs \geq 1 \rightarrow [\rho_2 \mapsto \lfloor zs \rfloor + 1]] = \Delta^{fix}_{insert}$$

Assume now the $\Delta_{20}$ of Example 5, which is obtained using $\Delta_{10}$. In order to make the example more legible we will omit $[\rho_2 \mapsto \ldots]$. By applying

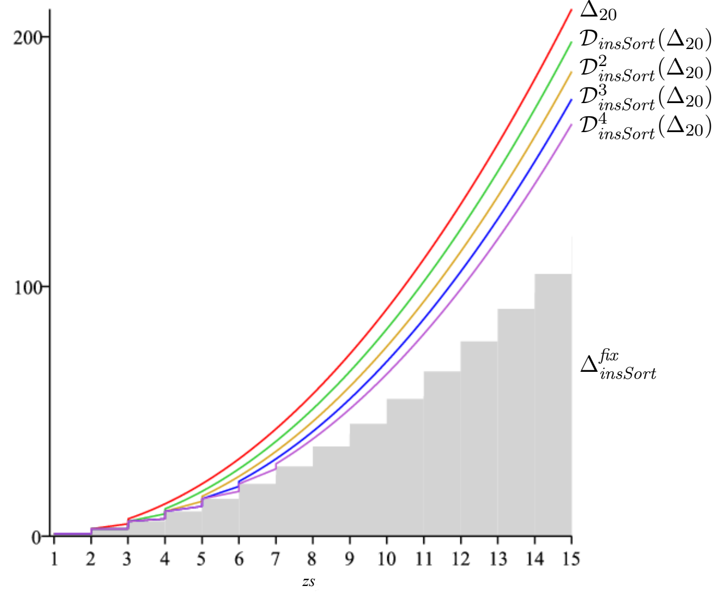



Figure 15: Graphical representation of $\mathcal{D}^i_{insSort}(\Delta_{20})$ $(i \in \{0..4\})$ and $\Delta^{fix}_{insSort}$.

$\mathcal{D}_{insSort}$ to this abstract heap we obtain:

$$\begin{array}{rcl}\mathcal{D}_{insSort}(\Delta_{20}) & = & [zs \geq 4 \rightarrow zs^2 - 2zs + 3] \sqcup \\ & & [zs \geq 3 \rightarrow 3zs - 3] \sqcup \\ & & [zs \geq 3 \rightarrow 3] \sqcup \\ & & [zs \geq 1 \rightarrow 1]\end{array}$$

which is strictly smaller than $\Delta_{20}$ from 3 onwards. Another iteration yields the following result:

$$\begin{array}{rcl}\mathcal{D}^2_{insSort}(\Delta_{20}) & = & [zs \geq 5 \rightarrow zs^2 - 3zs + 6] \sqcup \\ & & [zs \geq 4 \rightarrow 4zs - 6] \sqcup \\ & & [zs \geq 3 \rightarrow zs + 3] \sqcup \\ & & [zs \geq 2 \rightarrow 3] \sqcup \\ & & [zs \geq 1 \rightarrow 1]\end{array}$$

In Figure 15 we graphically represent $\Delta_{20}$ and iterations $\mathcal{D}_{insSort}(\Delta_{20})$ to $\mathcal{D}^4_{insSort}(\Delta_{20})$. In each iteration there is one more linear step which stabilises after one more iteration, that is the reason why in interval $[3,4)$ iterations $\mathcal{D}^2_{insSort}(\Delta_{20})$, $\mathcal{D}^3_{insSort}(\Delta_{20})$ and $\mathcal{D}^4_{insSort}(\Delta_{20})$ are equal; and in $[4,5)$ iterations $\mathcal{D}^3_{insSort}(\Delta_{20})$ and $\mathcal{D}^4_{insSort}(\Delta_{20})$ are equal.

The general form of the sequence is the following:

$$\begin{array}{rcl}\mathcal{D}^{i}_{insSort}(\Delta_{20}) & = & [zs \geq i+3 \rightarrow zs^2 - (i+1)zs + (i+1)(i+2)/2] \sqcup \\ & & [zs \geq i+2 \rightarrow (i+2)zs - (i+1)(i+2)/2] \sqcup \\ & & [zs \geq i+1 \rightarrow (i-1)zs + 3 - (i-2)(i-1)/2] \sqcup \\ & & [zs \geq i \rightarrow (i-2)zs + 3 - (i-3)(i-2)/2] \sqcup \\ & & \ldots \{j \in \{1..i-2\}\} \\ & & [zs \geq i-j \rightarrow (i-j-2)zs + 3 - (i-j-3)(i-j-2)/2] \sqcup \\ & & [zs \geq 1 \rightarrow 1]\end{array}$$

It converges to the function:

$$\Delta_{2\infty} = \left[zs \geq 2 \rightarrow (\lfloor zs \rfloor - 2)zs + 3 - \frac{(\lfloor zs \rfloor - 3)(\lfloor zs \rfloor - 2)}{2}\right] \sqcup [zs \geq 1 \rightarrow 1]$$

which on the natural numbers, where $zs = \lfloor zs \rfloor$, coincides with the fixpoint, (shown in Figure 15):

$$\Delta^{\mathit{fix}}_{insSort} = \left[zs \geq 1 \rightarrow \frac{\lfloor zs \rfloor (\lfloor zs \rfloor + 1)}{2}\right]$$

This fixpoint represents the exact heap consumption of the worst case of *insSort*.

In the following table we show the values of the first four iterations for naturals 1 to 7, and also the corresponding values of the fixpoint:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $\Delta_{20}$ | 1 | 3 | 7 | 13 | 21 | 31 | 43 |
| $\mathcal{D}_{insSort}(\Delta_{20})$ | 1 | 3 | 6 | 11 | 18 | 27 | 38 |
| $\mathcal{D}^{2}_{insSort}(\Delta_{20})$ | 1 | 3 | 6 | 10 | 16 | 24 | 34 |
| $\mathcal{D}^{3}_{insSort}(\Delta_{20})$ | 1 | 3 | 6 | 10 | 15 | 22 | 31 |
| $\mathcal{D}^{4}_{insSort}(\Delta_{20})$ | 1 | 3 | 6 | 10 | 15 | 21 | 29 |
| $\boldsymbol{\Delta^{\mathbf{fix}}_{\mathbf{insSort}}}$ | **1** | **3** | **6** | **10** | **15** | **21** | **28** |

Notice that, although the sequence converges to a strict upper bound of the fixpoint, each new iteration coincides with the fixpoint in one more natural number.

Had we used more precise functions as $\Delta^{\mathit{fix}}_{insert}$ and $nr_{insSort} = [zs \geq 1 \rightarrow \lfloor zs \rfloor - 1]$ we would have obtained:

$$\Delta'_{20} = [zs \geq 2 \rightarrow \lfloor zs \rfloor (\lfloor zs \rfloor - 1) + 1] \sqcup [zs \geq 1 \rightarrow 1]$$

which coincides with $\Delta_{20}$ on the natural numbers. In this case, we obtain a sequence of functions

$$\begin{array}{rcl} \mathcal{D}^i_{insSort}(\Delta'_{20}) & = & \left[zs \geq i+2 \rightarrow \lfloor zs \rfloor^2 - (i+1)\lfloor zs \rfloor + \frac{(i+1)(i+2)}{2}\right] \sqcup \\ & & \left[zs \geq 1 \rightarrow \frac{\lfloor zs \rfloor (\lfloor zs \rfloor + 1)}{2}\right] \end{array}$$

which converges to the fixpoint $\Delta^{fix}_{insSort}$. Each new iteration $i$ reaches the value of the fixpoint in the interval $(i+1, i+2)$. □

Notice that, in the *insert* function, the abstract heaps of every iteration are equal if we consider their domains only from a given threshold value. In some applications, it may suffice to obtain an expression $zs+1$ as an upper-bound to the costs of *insert*, even though it is not a fixed point of the corresponding iteration operator. This motivates the following definition.

**Definition 5.** Two abstract heaps $\Delta_1, \Delta_2 \in \mathbb{D}$ are said to be *asymptotically equivalent* (denoted $\Delta_1 \approx \Delta_2$) if there exists some $\overline{x_0}$ such that, for every $\overline{x}$ such that $\overline{x} \geq \overline{x_0}$, $\Delta_1(\overline{x}) = \Delta_2(\overline{x})$. The same applies to cost functions $\sigma, \mu \in \mathbb{F}$.

In our example above, it holds that $\Delta_{10} \approx \mathcal{D}_{insert}(\Delta_{10})$. In this case, we say that $\Delta_{10}$ is an *asymptotic fixed point* of $\mathcal{D}_{insert}$. We have already seen in Example 6 that $\Delta_2 \stackrel{\text{def}}{=} [zs \geq 1 \rightarrow [\rho_2 \mapsto \lfloor zs \rfloor + 1]]$ is an exact fixed point.

**Example 8.** Assume the $\Delta_{10}$ and $\mu_{10}$ of Example 5. By applying $\mathcal{M}_{\Delta_{10},insert}$ to $\mu_{10}$ we obtain:

$$\mathcal{M}_{\Delta_{10},insert}(\mu_{10}) = [zs \geq 3 \rightarrow zs+1] \sqcup [zs \geq 2 \rightarrow 3] \sqcup [zs \geq 1 \rightarrow 2]$$

which is a fixpoint of $\mathcal{M}_{\Delta_{10},insert}$. Notice that the operator depends on the $\Delta$ being used, and in this case we cannot improve $\mu$ further by iterating with the same $\Delta$. Consequently we could try iterating both $\Delta$ and $\mu$ at the same time. We use $\Delta_{1i}$ to denote $\mathcal{D}^i_{insert}(\Delta_{10})$ and $\mu_{1i}$ to denote $\mathcal{M}_{\Delta_{1i},insert}(\mu_{1(i-1)})$. Now we obtain the following sequence:

$$\begin{array}{rcl} \mu_{11} & = & [zs \geq 4 \rightarrow zs+1] \sqcup [zs \geq 3 \rightarrow 4] \sqcup [zs \geq 2 \rightarrow 3] \sqcup [zs \geq 1 \rightarrow 2] \\ \mu_{12} & = & [zs \geq 5 \rightarrow zs+1] \sqcup [zs \geq 4 \rightarrow 5] \sqcup [zs \geq 3 \rightarrow 4] \sqcup [zs \geq 2 \rightarrow 3] \sqcup \\ & & [zs \geq 1 \rightarrow 2] \\ \ldots & & \end{array}$$

Abbreviating:

$$\mu_{1i} = [zs \geq i+3 \to zs+1] \sqcup [zs \geq 1 \to \lfloor zs \rfloor + 1]$$

which converges to the fixpoint $\mu^{fix}_{insert} = [zs \geq 1 \to \lfloor zs \rfloor + 1]$. Again, this fixpoint represents the exact heap peak of the worst case of *insert*.

In order to calculate $\mu$ for *insSort* we have to determine which $\Delta$ and $\mu$ we assume for *insert*. Functions are analysed in dependency order, which means that we can assume we have already executed several iterations on $\Delta_{10}$ and $\mu_{10}$ before calculating $\Delta$ and $\mu$ for *insSort*. In this example, for simplicity, we will assume one iteration for each, i.e.

$$\begin{array}{lcl} \Delta_{11} & = & [zs \geq 3 \to zs+1] \sqcup [zs \geq 2 \to 3] \sqcup [zs \geq 1 \to 2] \\ \mu_{11} & = & [zs \geq 4 \to zs+1] \sqcup [zs \geq 3 \to 4] \sqcup [zs \geq 2 \to 3] \sqcup [zs \geq 1 \to 2] \end{array}$$

and in such case we calculate $\Delta'_{20}$ and $\mu'_{20}$:

$$\begin{array}{lcl} \Delta'_{20} & = & [zs \geq 4 \to zs^2 - zs + 1] \sqcup \\ & & [zs \geq 3 \to 3zs - 2] \sqcup \\ & & [zs \geq 2 \to 2zs - 1] \sqcup \\ & & [zs \geq 1 \to 1] \end{array}$$

$$\begin{array}{lcl} \mu'_{20} & = & [zs \geq 5 \to zs^2 - 2zs + 3] \sqcup \\ & & [zs \geq 4 \to 3zs - 1] \sqcup \\ & & [zs \geq 3 \to 2zs] \sqcup \\ & & [zs \geq 2 \to 3] \sqcup \\ & & [zs \geq 1 \to 1] \end{array}$$

Again if we iterate $\mu$ by fixing $\Delta$ we obtain immediately a fixpoint and we obtain poor results. If we iterate $\Delta$ and $\mu$ at the same time, we obtain a sequence of functions $\Delta'_{2i} = \mathcal{D}^i_{insSort}(\Delta'_{20})$ and $\mu'_{2i} = \mathcal{M}_{\Delta'_{2i}, insSort}(\mu'_{2(i-1)})$ such that for $i \geq 2$,

$$\begin{array}{lcl} \Delta'_{2i} & = & \mu'_{2(i-1)} \\ & = & [zs \geq i+4 \to zs^2 - (i+1)zs + (i+1)(i+2)/2] \sqcup \\ & & [zs \geq i+3 \to (i+3)zs + 1 - (i+2)(i+3)/2] \sqcup \\ & & [zs \geq i+2 \to (i+1)zs + 1 - i(i+1)/2] \sqcup \\ & & \ldots \{j \in \{-1..i-3\}\} \ldots \\ & & [zs \geq i-j \to (i-j-3)zs + 6 - (i-j-4)(i-j-3)/2] \sqcup \\ & & [zs \geq 2 \to 3] \sqcup \\ & & [zs \geq 1 \to 1] \end{array}$$

It converges to the function:

$$\begin{array}{rcl}\Delta'_{2\infty} & = & \mu'_{2\infty} \\ & = & [zs \geq 3 \to (\lfloor zs \rfloor - 3)zs + 6 - (\lfloor zs \rfloor - 4)(\lfloor zs \rfloor - 3)/2] \sqcup \\ & & [zs \geq 2 \to 3] \sqcup \\ & & [zs \geq 1 \to 1]\end{array}$$

which is strictly smaller than the $\Delta_{2\infty}$ obtained in Example 7, because we use more precise information about *insert*. It also coincides on the natural numbers with $\Delta^{fix}_{insSort}$, which is both a fixpoint of $\mathcal{D}_{insSort}$ and $\mathcal{M}_{\Delta^{fix}_{insSort}}$ when using $\Delta^{fix}_{insert}$ and $\mu^{fix}_{insert}$. This fixpoint represent the exact heap peak of the worst case of *insSort*.

If we concentrate only on the asymptotic part of the sequence $[zs \geq i + 4 \to zs^2 - (i+1)zs + (i+1)(i+2)/2]$ we can see that given a particular natural $zs \geq 4$, after $zs - 4$ iterations we get $\Delta'_{2(zs-4)}\ zs = 3 + \Delta^{fix}_{insSort}\ zs$.

Had we used $\Delta^{fix}_{insert}$, $\mu^{fix}_{insert}$ and $\Delta^{fix}_{insSort}$, then the initial approximation $\mu''_{20}$ would have already been the fixpoint $\mu^{fix}_{insSort}$. □

**Example 9.** Recall $\sigma_{10}$ in Example 5. By applying $\mathcal{S}_{insert}$ we obtain a sequence $\sigma_{1i} = \mathcal{S}^i_{insert}(\sigma_{10})$:

$$\begin{array}{rcl}\sigma_{1i} & = & [zs \geq i + 2 \to 8zs] \sqcup \\ & & [zs \geq i + 1 \to 8zs - 5] \sqcup \\ & & [zs \geq 1 \to 8 \lfloor zs \rfloor - 5]\end{array}$$

which converges to the fixpoint $\sigma^{fix}_{insert} = [zs \geq 1 \to 8 \lfloor zs \rfloor - 5]$.

If we assume $\sigma_{10}$ for *insert*, then we obtain $\sigma_{20}$ from Example 5. By applying $\mathcal{S}_{insSort}$ we obtain a sequence $\sigma_{2i} = \mathcal{S}^i_{insSort}(\sigma_{20})$ $(i \geq 2)$:

$$\begin{array}{rcl}\sigma_{2i} & = & [zs \geq i + 3 \to 14zs - (8i + 13)] \sqcup \\ & & [zs \geq i + 2 \to \sqcup\{8zs - 7, 14zs - (8i + 18)\}] \sqcup \\ & & [zs \geq 3 \to 8zs - 7] \sqcup \\ & & [zs \geq 2 \to \sqcup\{8zs - 12, 7\}] \sqcup \\ & & [zs \geq 1 \to 1]\end{array}$$

which converges to the function

$$\sigma_{2\infty} = [zs \geq 3 \to 8zs - 7] \sqcup [zs \geq 2 \to \sqcup\{8zs - 12, 7\}] \sqcup [zs \geq 1 \to 1]$$

Even in the natural numbers, this function is strictly bigger than the fixpoint $\sigma^{fix}_{insSort}$ when using $\sigma^{fix}_{insert}$ for *insert*:

$$\sigma^{fix}_{insSort} = [zs \geq 4 \to 8 \lfloor zs \rfloor - 12] \sqcup [zs \geq 1 \to 6 \lfloor zs \rfloor - 5]$$

The latter is obtained as limit of the sequence generated when using $\sigma^{fix}_{insert}$ and $len_{insSort} = \lfloor zs \rfloor$. This fixpoint represents the exact stack consumption of the worst case of *insSort*. $\square$

From the previous examples we have learned some useful lessons. First, simultaneous iteration of $\Delta$ and $\mu$ produces better results than iterating $\mu$ with a fixed $\Delta$. Second, succesive iterations may lead to a strictly decreasing infinite sequence in which each iteration strictly improves the previous bound but also generates a more complex expression, i.e. with more guarded expressions and upper bounds which cannot be removed by simplification. We expect to obtain automatically neither the general term of the sequence nor its limit, but the graphical representation of the iterations helps an interested user to guess them, and then maybe even to obtain a fixpoint. If we are only interested in the asymptotic costs we can get rid of the lower part of the functions, which makes expressions simpler, and iterate on the upper part to obtain again a new asymptotic bound, or an asymptotic fixpoint. Third, when analysing a function which depends on others, the better signatures we have for them the better results we will obtain for the new function. Additionally, the accuracy of our bounds strongly depends on the accuracy of the external analyses (i.e. size analysis, *nb*, *nr* and *len*).

### *3.4. Reductivity in absence of admissibility conditions*

As we have seen, reductivity only holds under some admissibility conditions on the externally given *nb*, *nr*, and *len* functions. If one of these admissibility conditions does not hold, then the initial signature may not be reductive.

Given the above, let us assume that $(\Delta_0, \mu_0, \sigma_0)$ is correct, but not reductive. If we define, $\Delta_1 = \mathcal{D}_f(\Delta_0)$, $\mu_1 = \mathcal{M}_{\Delta_0,f}(\mu_0)$ and $\sigma_1 = \mathcal{S}_f(\sigma_0)$, we know, by the correctness property of the abstract interpretation [16], that the new signature $(\Delta_1, \mu_1, \sigma_1)$ is correct, but not necessarily more precise than $(\Delta_0, \mu_0, \sigma_0)$. If $\Delta_1 \sqsupset \Delta_0$ then we can safely discard $\Delta_1$, since our initial approximation $\Delta_0$ is more precise. It could also be the case that $\Delta_1$ and $\Delta_0$ are not comparable, in which case we compute $\Delta'_1 = \sqcap\{\Delta_1, \Delta_0\}$, since it is more precise than both $\Delta_1$ and $\Delta_0$, but still correct. The same reasoning applies to the $\mu_0$ and $\sigma_0$ components.

Therefore, in absence of the reductivity property we can define the fol-

lowing modified iteration operators as follows:

$$\begin{array}{rcl}\mathcal{D}'_f(\Delta) & = & \sqcap\{\Delta, \mathcal{D}_f(\Delta)\} \\ \mathcal{M}'_{\Delta,f}(\mu) & = & \sqcap\{\mu, \mathcal{M}_{\Delta,f}(\mu)\} \\ \mathcal{S}'_f(\sigma) & = & \sqcap\{\sigma, \mathcal{S}_f(\sigma)\}\end{array}$$

for every $\Delta \in \mathbb{D}$, $\mu \in \mathbb{F}$ and $\sigma \in \mathbb{F}$. Obviously, $\mathcal{D}'_f(\Delta) \sqsubseteq \Delta$ and similarly for the other two operators. So, trivially, if $(\Delta_0, \mu_0, \sigma_0)$ is our initial signature,

$$\Delta_0 \in \mathit{Red}(\mathcal{D}'_f) \quad \wedge \quad \mu_0 \in \mathit{Red}(\mathcal{M}'_{\Delta_0,f}) \quad \wedge \quad \sigma_0 \in \mathit{Red}(\mathcal{S}'_f)$$

and, for every $n \geq 0$, $((\mathcal{D}'_f)^n(\Delta_0), (\mathcal{M}'_{\Delta_0,f})^n(\mu_0), (\mathcal{S}'_f)^n(\sigma_0))$ is a correct signature.

## 4. Case studies

In this section we apply our space analysis to several examples. Some of the functions have already been introduced in previous sections. All the algorithms for inferring costs have been implemented in *Maple*. The compiler's front-end generates a representation of the abstract syntax tree corresponding to the program being analysed, and *Maple* computes the initial symbolic approximations and perform the necessary simplifications. This system is also used for computing the asymptotic expressions to the obtained bounds.

**Example 10.** Let us consider the problem of finding the longest increasing subsequence (LIS) of a list given as parameter. For instance, the list $[12, 2, 3, 9, 5, 7, 1]$ has $[2, 3, 5, 7]$ as its LIS. In general, assume a list $[x_1, \ldots, x_n]$. The idea relies in building a list of $n$ pairs $[(l_1, ms_1), \ldots, (l_n, ms_n)]$, where each $l_j$ describes the length of the LIS starting with the element $x_j$, and $ms_j$ is the LIS itself. The generation of this list is performed in reverse order. We start with the last element $x_n$ of the input list, whose corresponding pair is, obviously, $(1, [x_n])$. Assume we have computed the pairs $[(l_{i+1}, ms_{i+1}), \ldots, (l_n, ms_n)]$ corresponding to the last $n - i$ elements of the list. The LIS that starts from $x_i$ can be done by taking the maximum among all the $l_j$ (being $j \in i + 1..n$) such that the head of the LIS that starts from $x_j$ (that is, $ms_j$) is strictly greater than $x_i$. Once we have computed all the pairs, the LIS of the whole list is determined by the pair with the highest first component.

```
maximum :: [(Int, α)@ρ1]@ρ2 → (Int, α)@ρ1
maximum [p] = p
maximum (p : xs) | l ≥ l' = p
                 | otherwise = p'
    where {(l, _) = p; (l', _) = p'; p' = maximum xs}

filterLower :: Int → [(Int, [Int]@ρ1)@ρ2]@ρ3 → ρ4 → [(Int, [Int]@ρ1)@ρ2]@ρ4
filterLower x [ ] @ r4 = [ ] @ r4
filterLower x (p : xs) @ r4
    | x < head ms   = (p : filterLower x xs @ r4)@r4
    | otherwise     = filterLower x xs @ r4
    where (_, ms) = p


lis :: [Int]@ρ1 → ρ2 → ρ3 → ρ4 → [(Int, [Int]@ρ2)@ρ3]@ρ4
lis [ ] @ r2 r3 r4 = [ ] @ r4
lis (x : xs) @ r2 r3 r4
    | null lxs'     = (((1, x : [ ]@r2)@r3) : lxs)@r4
    | otherwise     = let (l, ms) = maximum lxs'
                      in (((1 + l, (x : ms)@r2)@r3) : lxs)@r4
  where  lxs    = lis xs @ r2 r3 r4
         lxs'   = filterLower x lxs @ self

lis' :: [Int]@ρ1 → ρ2 → ρ3 → (Int, [Int]@ρ2)@ρ3
lis' xs @ r2 r3 = maximum (lis xs @ r2 r3 self)
```

Figure 16: *Full-Safe* code (with regions) of the algorithm finding the LIS.

In Figure 16 we show the *Full-Safe* code, being annotated with regions by the compiler. The *lis* function is given the input list $[x_1, \ldots, x_n]$, and computes the list with the above mentioned pairs. The recursive call to *lis* computes the list *lxs* of pairs corresponding to the the tail of the input. The *filterLower* function discards all the pairs whose associated LIS starts with an element lower or equal than the head of the input. Among the remaining pairs, the *maximum* function selects the pair $(l_j, ms_j)$ with the highest $l_j$. Finally, the *lis'* function selects the maximal pair among those computed by *lis*, which contains the length of the LIS, and the LIS itself.

Let us assume that the $\Sigma$ environment already contains the heap cost of *maximum* and *filterLower*, so that $\Sigma(\mathit{maximum}) = (\Delta_{max}, \mu_{max}, _)$ and $\Sigma(\mathit{filterLower}) = (\Delta_{filt}, \mu_{filt}, _)$, being $\Delta_{max}$, $\Delta_{filt}$, $\mu_{max}$, and $\mu_{filt}$ defined as

follows:

$$\begin{array}{rcl}\Delta_{max} & = & [xs \geq 1 \rightarrow [\,]] \\ \mu_{max} & = & [xs \geq 1 \rightarrow 0]\end{array}$$

$$\begin{array}{rcl}\Delta_{filt} & = & [xs \geq 1 \rightarrow 1] \sqcup [xs \geq 2 \rightarrow xs] \\ \mu_{filt} & = & [xs \geq 1 \rightarrow 1] \sqcup [xs \geq 2 \rightarrow 2] \sqcup [xs \geq 3 \rightarrow xs]\end{array}$$

All these results are asymptotic fixed points. Now we apply *computeDelta* to the *lis* function under this environment $\Sigma$. The partial results given for each region are the following:

$$\begin{array}{rclrcl}\Delta_{b,lis}\,\rho_2 & = & [xs \geq 1 \rightarrow 0] & \Delta_{r,lis}\,\rho_2 & = & [xs \geq 2 \rightarrow 2] \\ \Delta_{b,lis}\,\rho_3 & = & [xs \geq 1 \rightarrow 0] & \Delta_{r,lis}\,\rho_3 & = & [xs \geq 2 \rightarrow 1] \\ \Delta_{b,lis}\,\rho_4 & = & [xs \geq 1 \rightarrow 1] & \Delta_{r,lis}\,\rho_4 & = & [xs \geq 2 \rightarrow 1] \\ \Delta_{b,lis}\,\rho_{self} & = & [xs \geq 1 \rightarrow 0] & \Delta_{r,lis}\,\rho_{self} & = & [xs \geq 2 \rightarrow 1] \sqcup \\ & & & & & [xs \geq 3 \rightarrow xs - 1]\end{array}$$

Assume that $nb_{lis} = [xs \geq 1 \rightarrow 1]$ and that $nr_{lis} = [xs \geq 1 \rightarrow xs - 1]$. The *computeDelta* algorithm yields the following result:

$$\begin{array}{rcl}\Delta_{0,lis}\,\rho_2 & = & [xs \geq 1 \rightarrow 0] \sqcup [xs \geq 2 \rightarrow 2xs - 2] \\ \Delta_{0,lis}\,\rho_3 & = & [xs \geq 1 \rightarrow 0] \sqcup [xs \geq 2 \rightarrow xs - 1] \\ \Delta_{0,lis}\,\rho_4 & = & [xs \geq 1 \rightarrow 1] \sqcup [xs \geq 2 \rightarrow xs]\end{array}$$

By iterating the $[\![\cdot]\!]_\Delta$ interpretation on this result we get $\Delta_{1,lis} = \mathcal{D}_{lis}(\Delta_{0,lis})$, where the charges done in $\rho_2$ specified by $\Delta_{1,lis}$ are as follows:

$$\Delta_{1,lis}\,\rho_2 \quad = \quad [xs \geq 1 \rightarrow 0] \sqcup [xs \geq 2 \rightarrow 2] \sqcup [xs \geq 3 \rightarrow 2xs - 2]$$

This differs from $\Delta_{0,lis}\,\rho_2$ only in the interval $[2, 3)$. Another iteration of $\mathcal{D}_{lis}$ yields the following result:

$$\begin{array}{rcl}\Delta_{2,lis}\,\rho_2 & = & [xs \geq 1 \rightarrow 0] \sqcup [xs \geq 2 \rightarrow 2] \sqcup \\ & & [xs \geq 3 \rightarrow 4] \sqcup [xs \geq 4 \rightarrow 2xs - 2]\end{array}$$

which differs from the previous one in the interval $[3, 4)$. Since these iterations are asymptotically equivalent to $\Delta_{0,lis}$, we give $\Delta_{0,lis}$ as parameter to *computeMu*, obtaining the following result:

$$\mu_{0,lis} \quad = \quad [xs \geq 1 \rightarrow 1] \sqcup [xs \geq 2 \rightarrow 6] \sqcup [xs \geq 3 \rightarrow 5xs - 4]$$

$$
\begin{array}{l}
\textit{insertT}\ y\ \textit{Empty} \mathbin{@} r_1 = \textit{Node}\ (\textit{Empty} \mathbin{@} r_1)\ y\ (\textit{Empty} \mathbin{@} r_1) \mathbin{@} r_1 \\
\textit{insertT}\ y\ (\textit{Node}\ l\ x\ r) \mathbin{@} r_1 \\
\qquad\qquad \mid\ x == y \quad = \textit{Node}\ l\ x\ r \mathbin{@} r_1 \\
\qquad\qquad \mid\ y < x \quad\ \ = \textit{Node}\ (\textit{insertT}\ y\ l \mathbin{@} r_1)\ x\ r \mathbin{@} r_1 \\
\qquad\qquad \mid\ y > x \quad\ \ = \textit{Node}\ l\ x\ (\textit{insertT}\ y\ r \mathbin{@} r_1) \mathbin{@} r_1 \\
\\
\textit{mkTree}\ [\,] \mathbin{@} r_2 = \textit{Empty} \mathbin{@} r_2 \\
\textit{mkTree}\ (x : xs) \mathbin{@} r_2 = \textit{insertT}\ x\ (\textit{mkTree}\ xs \mathbin{@} r_2) \mathbin{@} r_2 \\
\\
\textit{inorderAcc}\ \textit{Empty}\ xs \mathbin{@} r_2 = xs \\
\textit{inorderAcc}\ (\textit{Node}\ l\ x\ r)\ xs \mathbin{@} r_2 = \textit{inorderAcc}\ l\ (x : rs)@r_2 \mathbin{@} r_2 \\
\qquad \textbf{where}\ rs = \textit{inorderAcc}\ r\ xs \mathbin{@} r_2 \\
\\
\textit{inorder}\ t \mathbin{@} r_2 = \textit{inorderAcc}\ t\ [\,]@r_2 \mathbin{@} r_2 \\
\\
\textit{treeSort}\ xs \mathbin{@} r_2 = \textit{inorder}\ (\textit{mkTree}\ xs \mathbin{@} \textit{self}) \mathbin{@} r_2
\end{array}
$$

Figure 17: Region inference for the treesort algorithm.

Now we apply the $\mathcal{M}_{lis}$ operator with the previously obtained abstract heaps. Let $\mu_{1,lis} = \mathcal{M}_{\Delta_{1,lis},lis}(\mu_{0,lis})$ and $\mu_{2,lis} = \mathcal{M}_{\Delta_{2,lis},lis}(\mu_{1,lis})$. We get:

$$
\begin{array}{rcl}
\mu_{1,lis} & = & [xs \geq 1 \to 1] \sqcup [xs \geq 2 \to 6] \sqcup [xs \geq 3 \to xs + 8] \sqcup \\
 & & [xs \geq 4 \to 5xs - 4] \\
\mu_{2,lis} & = & [xs \geq 1 \to 1] \sqcup [xs \geq 2 \to 6] \sqcup [xs \geq 3 \to xs + 8] \sqcup \\
 & & [xs \geq 4 \to xs + 12] \sqcup [xs \geq 5 \to 5xs - 4]
\end{array}
$$

In this case, we already reach a fixed point in the first iteration. It is interesting to note that, even if each call to *lis* does charges in its *self* region, these are not reflected in $\Delta_{self}$ of *computeMu*, as they are done *after* the recursive call to *lis*. These charges are included in the $\mu_{aft,lis}$ component, which is not multiplied by the length of the longest call chain $len_{lis}$. As a consequence, we get a linear bound, instead of a quadratic one. □

The assignment of size functions to the variables of the function is straightforward in many programs involving list manipulations. However, this task may be more involved when working with programs that manipulate nonlinear data structures, such as binary trees.

**Example 11.** Recall the *tree sort* algorithm of Example 2. The *Full-Safe* code of the whole algorithm is shown in Figure 17. Let us start with $insertT$. Assuming that the tree given as parameter has size $t$, we must determine the sizes of its left and right subtrees, pointed to by the variables $l$ and $r$, respectively. The root of the tree takes one cell, so the sum of the sizes of the subtrees must be equal to $t-1$. Moreover, and since the smallest tree we can build (*Empty*) needs one cell, we know that the sizes of the subtrees must range from 1 to $t-2$. If we want to be on the safe side, we can assume that *both* trees are of size $t-2$, which is a correct approximation to their actual runtime sizes. Hence we get the following size functions:

$$|l| = t-2 \qquad |r| = t-2 \tag{2}$$

With respect to the call-tree information, we have to assume the worst case, in which $t$ is a degenerate tree. In this case, $insertT$ does as many recursive calls as the number of recursive nodes, and a single base call. Every binary tree of size $t$ is made up of $(t-1)/2$ cells with the *Node* constructor, and $(t+1)/2$ cells with the *Empty* constructor[2]. Hence we get:

$$nb_{insertT} = 1 \qquad nr_{insertT} = \frac{t-1}{2} \qquad len_{insertT} = \frac{t+1}{2}$$

The *computeDelta* algorithm yields the following initial approximation:

$$\Delta_{0,insertT} = [t \geq 1 \to 3] \sqcup [t \geq 3 \to (t+5)/2]$$

which is an asymptotic fixed point. We use this bound for computing the successive $\mu$ approximations,

$$\begin{array}{rcl}
\mu_{0,insertT} & = & [t \geq 1 \to 3] \sqcup [t \geq 3 \to 6] \sqcup [t \geq 5 \to (t+9)/2] \\
\mu_{1,insertT} & = & [t \geq 1 \to 3] \sqcup [t \geq 3 \to 4] \sqcup [t \geq 5 \to 6] \\
& & \sqcup [t \geq 5 \to (t+7)/2] \\
\mu_{2,insertT} & = & [t \geq 1 \to 3] \sqcup [t \geq 3 \to 4] \sqcup [t \geq 5 \to 6] \\
& & \sqcup [t \geq 5 \to (t+5)/2]
\end{array}$$

the last of which is an asymptotic fixed point. Now we assume that we apply the *mkTree* function to a list of size $zs$. We start from the following size information, given externally:

$$|xs| = zs - 1 \qquad |mkTree\ xs| = 2zs - 3$$

[2]The number of cells taken by a binary tree is always an odd number, so there is no need to truncate the results of dividing by two.

The latter result is due to the fact that *mkTree* converts a list with $n$ elements (that is, of size $n+1$) into a tree made of $n$ cells with the *Node* constructor, and $n+1$ cells with the *Empty* constructor. These amount to $2n+1$ cells. Since a list of size $xs$ has $xs-1$ elements, the size of the result of *mkTree* $xs$ is $2(xs-1)+1 = 2xs-1$ or, as a function on $zs$, $2(zs-1)-1 = 2zs-3$. Moreover, we assume $nb_{mkTree} = 1$, $nr_{mkTree} = zs-1$, and $len_{mkTree} = zs$. We obtain the following sequence of asymptotic bounds:

$$\begin{array}{lcl} \Delta_{0,mkTree} & \approx & zs^2 \\ \Delta_{1,mkTree} & \approx & zs^2 - zs + 2 \\ \Delta_{2,mkTree} & \approx & zs^2 - 2zs + 5 \\ \Delta_{3,mkTree} & \approx & zs^2 - 3zs + 9 \end{array}$$

all of them correspond to the charges done to $\rho_2$. None of these functions is an asymptotic fixed point. With respect to the heap needs, the *computeMu* function yields $\mu_{0,mkTree} \approx zs^2 - 4zs + 14$, which is an asymptotic fixed point.

A call to *inorderAcc* $t$ does as many recursive calls as internal nodes in the input tree $t$, and as many base calls as *Empty* leaves, so $nr_{inorderAcc} = (t-1)/2$, and $nb_{inorderAcc} = (t+1)/2$. Regarding the call-tree height, we consider the worst-case of a degenerate tree with $len_{inorderAcc} = (t+1)/2$. Assigning sizes to the $l$ and $r$ variables in *inorderAcc* is more involved. If we proceeded as in (2), we would obtain the following bound to the charges done to $\rho_2$:

$$\Delta_{0,inorderAcc} = [t \geq 1 \rightarrow 0] \sqcup [t \geq 3 \rightarrow (t-1)/2] \tag{3}$$

However, our $nb_{inorderAcc}$, $nr_{inorderAcc}$, and $len_{inorderAcc}$ functions would not be admissible w.r.t. *inorderAcc*, so we could not ensure reductivity of $\Delta_{0,inorderAcc}$ w.r.t. $\mathcal{D}_{inorderAcc}$. Another possibility is to leave $|l|$ unspecified, and define $|r| = t - (|l|\ t\ xs) - 1$. In this case we also obtain the $\Delta_{0,inorderAcc}$ of (3), since the $|l|\ t\ xs$ terms are canceled out in the result. However, these terms appear again when applying the $\mathcal{D}$ operator, which leads to unintelligible bounds that depend on the size of $|l|$. In this particular example we can take advantage of the fact that the heap costs of *inorderAcc* do not depend on how well-balanced is its input tree, so we can safely assume a degenerate tree and assign $|l| = 1$, so $|r| = t-2$. In this case we also get (3) as a result, and we can iterate the abstract interpretation in order to get:

$$\Delta_{1,inorderAcc} = [t \geq 1 \rightarrow 0] \sqcup [t \geq 3 \rightarrow 1] \sqcup [t \geq 5 \rightarrow (t-1)/2]$$

Hence, $\Delta_{0,inorderAcc}$ is an asymptotic fixed point, from which we obtain $\mu_0 \approx (t-1)/2$. Finally, we can apply the abstract interpretation function on *treeSort*, which yields the following results:

$$\Delta_{treeSort} \approx \left[ \begin{array}{lll} \rho_2 & \mapsto & xs \\ \rho_{self} & \mapsto & xs^2 - 3xs + 9 \end{array} \right] \quad \mu_{treeSort} \approx xs^2 - 2xs + 10$$

□

**Example 12.** In Figures 18, 19 and 20 we show the results of applying our memory consumption analysis to several example functions, most of them for processing lists. The values $\Delta_0$, $\mu_0$ and $\sigma_0$ represent the initial upper bounds obtained by the *computeDelta*, *computeMu* and *computeSigma* functions. In case these bounds are not fixed points of their corresponding operators $\mathcal{D}$, $\mathcal{M}$ and $\mathcal{S}$, we represent the results of successive applications of these operators. When $i > 0$, $\Delta_i$ denotes the abstract heap $\mathcal{D}^i(\Delta_0)$. The same applies with the $\mu_i$ and $\sigma_i$. In the case of $\mu_i$, the $i$-th application of the $\mathcal{M}$ operator is performed with the corresponding $\Delta_i$. A $(\star)$ mark attached to an abstract heap or cost function indicates that it is an asymptotic fixed point of its corresponding operator.

Some of the functions have already been introduced in this paper: *append* and *appendC* (Example 1). The *msort* function implements the *Mergesort* algorithm, and it depends on *split* and *merge*. The former obtains the first $n$ elements of the list $xs$ given as input, whereas the latter merges two sorted lists into a single one. The *partition* and *qsort* functions make up the standard implementation of *Quicksort* algorithm.

A call to *pascal* $n$ computes the $n$-th row of the Pascal triangle by using an auxiliary function *sumList* which transforms the list $[x_1, x_2, \ldots, x_n]$ (corresponding to the $n$-th row of the Pascal triangle) into the list $[x_1 + x_2, x_2 + x_3, \ldots, x_n]$ (containing the elements of the $n+1$-th row, excluding the first 1). The *combNumbers* function uses the result of *pascal* in order to compute the binomial coefficient of the numbers given as parameters. The *fib* function implements the following naive, memory consuming approach of computing Fibonacci numbers, in which numbers given as parameters are stored in the heap.

In some cases, we would get an infinite descending chain $\mu_0 \sqsupset \mu_1 \sqsupset \ldots \sqsupset \mu_i \sqsupset \ldots$ of functions, as in the $\mu$ component of *appendC*. However, this does not mean that the limit of this chain when $i$ tends to $+\infty$ is the zero-constant

*append xs ys @ $\rho_3$*

$$\Delta_0 \approx [\rho_2 \mapsto xs - 1]^{(\star)}$$

$$\mu_0 \approx xs - 1^{(\star)}$$

$$\sigma_0 \approx 7xs^{(\star)}$$

*appendC xs ys @ $\rho_3$*

$$\Delta_0 \approx [\rho_3 \mapsto xs + ys - 1]^{(\star)}$$

$$\begin{aligned} \mu_0 &\approx 2ys + xs - 2 \\ \mu_1 &\approx 2ys + xs - 3 \\ \mu_2 &\approx 2ys + xs - 4 \end{aligned}$$

$$\sigma_0 \approx 7xs^{(\star)}$$

*length xs*

$$\Delta_0 \approx [\,]^{(\star)}$$

$$\mu_0 \approx 0$$

$$\sigma_0 \approx 5xs^{(\star)}$$

*split n xs @ $\rho_1$ $\rho_2$ $\rho_3$*

$$\Delta_0 \approx \begin{bmatrix} \rho_1 \mapsto 1 \\ \rho_2 \mapsto \min(n, xs + 1) + 1 \\ \rho_3 \mapsto \min(n, xs + 1) + 1 \end{bmatrix}^{(\star)}$$

$$\begin{aligned} \mu_0 &\approx 6 + 2\min(xs, n - 1) \\ \mu_1 &\approx 5 + 2\min(xs, n - 1)^{(\star)} \end{aligned}$$

$$\sigma_0 \approx 9\min(n + 1, xs + 2) + 1^{(\star)}$$

*merge xs ys @ $\rho_1$*

$$\Delta_0 \approx [\rho_1 \mapsto 2xs + 2ys - 3]^{(\star)}$$

$$\mu_0 \approx 2xs + 2ys - 3^{(\star)}$$

$$\sigma_0 \approx 11xs + 11ys - 10^{(\star)}$$

*msort xs @ $\rho_1$ $\rho_2$*

$$\begin{aligned}
\Delta_0 &\approx \begin{bmatrix} \rho_1 \mapsto \frac{1}{2}xs^2 + \frac{1}{2}xs - 3 \\ \rho_2 \mapsto 2xs^2 - 3xs \end{bmatrix} \\
\Delta_1 &\approx \begin{bmatrix} \rho_1 \mapsto \frac{1}{4}xs^2 + \frac{3}{2}xs - \frac{15}{4} \\ \rho_2 \mapsto xs^2 + xs - 3 \end{bmatrix} \\
\Delta_2 &\approx \begin{bmatrix} \rho_1 \mapsto \frac{1}{8}xs^2 + \frac{9}{4}xs - \frac{35}{8} \\ \rho_2 \mapsto \frac{1}{2}xs^2 + 4xs - \frac{11}{2} \end{bmatrix} \\
\Delta_3 &\approx \begin{bmatrix} \rho_1 \mapsto \frac{1}{16}xs^2 + \frac{23}{8}xs - \frac{79}{16} \\ \rho_2 \mapsto \frac{1}{4}xs^2 + \frac{13}{2}xs - \frac{31}{4} \end{bmatrix} \\
\Delta_4 &\approx \begin{bmatrix} \rho_1 \mapsto \frac{1}{32}xs^2 + \frac{55}{16}xs - \frac{175}{32} \\ \rho_2 \mapsto \frac{1}{8}xs^2 + \frac{35}{4}xs - \frac{79}{8} \end{bmatrix}
\end{aligned}$$

$$\begin{aligned}
\mu_0 &\approx \tfrac{15}{8}xs^2 + \tfrac{1}{2}xs\log(xs - 1) + 2xs \\ &\quad + \tfrac{1}{2}\log(xs - 1) - \tfrac{55}{8} \\
\mu_1 &\approx \tfrac{25}{32}xs^2 + \tfrac{1}{4}xs\log(xs - 1) + \tfrac{73}{16}xs \\ &\quad + \tfrac{3}{4}\log(xs - 1) - \tfrac{299}{32} \\
\mu_2 &\approx \tfrac{45}{128}xs^2 + \tfrac{1}{8}xs\log(xs - 1) + \tfrac{447}{64}xs \\ &\quad + \tfrac{7}{8}\log(xs - 1) - \tfrac{1579}{128} \\
\mu_3 &\approx \tfrac{85}{512}xs^2 + \tfrac{1}{16}xs\log(xs - 1) + \tfrac{2419}{256}xs \\ &\quad + \tfrac{15}{16}\log(xs - 1) - \tfrac{7995}{512} \\
\mu_4 &\approx \tfrac{165}{2048}xs^2 + \tfrac{1}{32}xs\log(xs - 1) \\ &\quad + \tfrac{12235}{1024}xs + \tfrac{31}{32}\log(xs - 1) - \tfrac{38971}{2048}
\end{aligned}$$

$$\begin{aligned} \sigma_0 &\approx 14\log(xs - 1) + 11xs + 15 \\ \sigma_1 &\approx 11xs + 1^{(\star)} \end{aligned}$$

*partition y xs @ $\rho_2$ $\rho_3$ $\rho_4$*

$$\Delta_0 \approx \begin{bmatrix} \rho_2 \mapsto xs \\ \rho_3 \mapsto xs \\ \rho_4 \mapsto xs \end{bmatrix}^{(\star)}$$

$$\begin{aligned} \mu_0 &\approx 3xs \\ \mu_1 &\approx 3xs - 1^{(\star)} \end{aligned}$$

$$\sigma_0 \approx 9xs - 2^{(\star)}$$

Figure 18: Results of the space analysis (1)

$qsort\ xs\ @\ \rho_1\ \rho_2$

$$\Delta_0 \approx \begin{bmatrix} \rho_1 \mapsto 2xs^2 - 4xs + 2 \\ \rho_2 \mapsto xs^2 - xs + 1 \end{bmatrix}$$
$$\Delta_1 \approx \begin{bmatrix} \rho_1 \mapsto 2xs^2 - 6xs + 6 \\ \rho_2 \mapsto xs^2 - 2xs + 3 \end{bmatrix}$$
$$\Delta_2 \approx \begin{bmatrix} \rho_1 \mapsto 2xs^2 - 8xs + 12 \\ \rho_2 \mapsto xs^2 - 3xs + 6 \end{bmatrix}$$
$$\Delta_3 \approx \begin{bmatrix} \rho_1 \mapsto 2xs^2 - 10xs + 20 \\ \rho_2 \mapsto xs^2 - 4xs + 10 \end{bmatrix}$$

$\mu_0 \approx 7xs^2 - 19xs + 19$
$\mu_1 \approx 7xs^2 - 30xs + 42$
$\mu_2 \approx 7xs^2 - 41xs + 76$
$\mu_3 \approx 7xs^2 - 52xs + 121$

$\sigma_0 \approx 20xs - 9$
$\sigma_1 \approx 20xs - 19$
$\sigma_2 \approx 20xs - 29$
$\sigma_3 \approx 20xs - 39$

$sumList\ xs\ @\ \rho_2$

$\Delta_0 \approx [\rho_2 \mapsto xs]^{(\star)}$

$\mu_0 \approx xs + 1$
$\mu_1 \approx xs^{(\star)}$

$\sigma_0 \approx 9xs - 9^{(\star)}$

$combNumbers\ m\ n$

$\Delta_0 \approx [\,]^{(\star)}$

$\mu_0 \approx m^2 - m + 12^{(\star)}$

$\sigma_0 \approx 15m + 17^{(\star)}$

$unshuffle\ xs\ @\ \rho_2\ \rho_3$

$$\Delta_0 \approx \begin{bmatrix} \rho_2 \mapsto xs + 1 \\ \rho_3 \mapsto xs \end{bmatrix}^{(\star)}$$

$\mu_0 \approx 2xs + 2$
$\mu_1 \approx 2xs + 1^{(\star)}$

$\sigma_0 \approx 7xs + 2^{(\star)}$

$pascal\ n\ @\ \rho_1$

$\Delta_0 \approx [\rho_1 \mapsto n^2 + 3n + 2]$
$\Delta_1 \approx [\rho_1 \mapsto n^2 + 2n + 3]$
$\Delta_2 \approx [\rho_1 \mapsto n^2 + n + 5]$
$\Delta_3 \approx [\rho_1 \mapsto n^2 + 8]$

$\mu_0 \approx n^2 + 2n + 3$
$\mu_1 \approx n^2 + n + 5$
$\mu_2 \approx n^2 + 8$
$\mu_3 \approx n^2 - n + 12$

$\sigma_0 \approx 15n + 13$
$\sigma_1 \approx 15n + 4$
$\sigma_2 \approx 15n - 5$
$\sigma_3 \approx 15n - 14$

$fib\ n\ @\ \rho_1$

$\Delta_0 \approx [\rho_1 \mapsto 2^n - 1]$
$\Delta_1 \approx [\rho_1 \mapsto 2^{n-1} + 2^{n-2} - 1]$
$\Delta_2 \approx [\rho_1 \mapsto 2^{n-1} + 2^{n-4} - 1]$

$$\Delta_3 \approx [\rho_1 \mapsto 2^{n-3} + 3 \cdot 2^{n-4} + 3 \cdot 2^{n-5} + 2^{n-6} - 1]$$

$\mu_0 \approx 4 \cdot 2^{n-2} + 2 \cdot 2^{n-3} - 3$
$\mu_1 \approx 4 \cdot 2^{n-3} + 6 \cdot 2^{n-4} + 2 \cdot 2^{n-5} - 4$
$\mu_2 \approx 2 \cdot 2^{n-4} + 6 \cdot 2^{n-5} + 8 \cdot 2^{n-6} + 2 \cdot 2^{n-7} - 5$
$\mu_3 \approx 2 \cdot 2^{n-5} + 8 \cdot 2^{n-6} + 12 \cdot 2^{n-7} + 10 \cdot 2^{n-8} + 2 \cdot 2^{n-9} - 7$

$\sigma_0 \approx 5n + 7$
$\sigma_1 \approx 5n + 6$
$\sigma_2 \approx 5n + 5$
$\sigma_2 \approx 5n + 4$

Figure 19: Results of the space analysis (2)

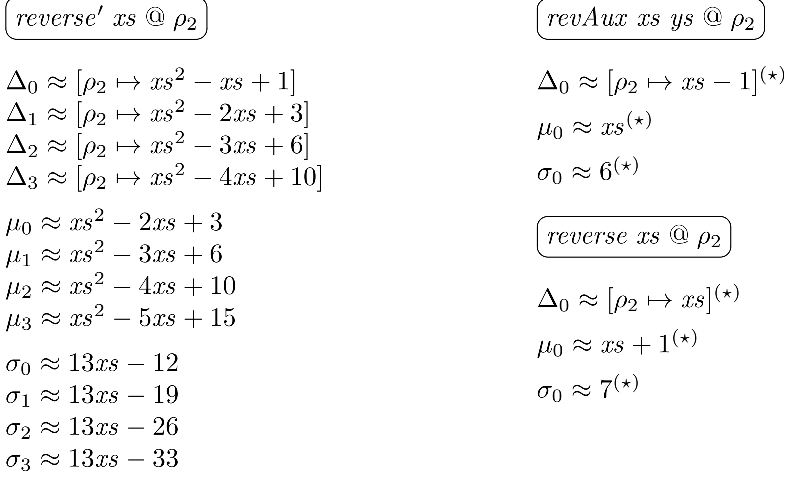



Figure 20: Results of the space analysis (3)

function, since the results shown here are only asymptotic bounds, and the limit does not have to coincide asymptotically with the $\mu_i$. In the case of *appendC*, the sequence $\{\mu_i\}_{i\in\mathbb{N}}$ converges pointwise to $xs + ys - 1$, which is a fixed point of the $\mathcal{M}$ operator. With the stack costs of *qsort* we get a similar situation. Normally, the initial bounds are overapproximations of the actual fixed points. In the case of *msort*, whose worst-case heap space complexity is in $\mathcal{O}(xs \log\, xs)$, we get a quadratic bound. Notice, however, that the $xs^2$ coefficient keeps decreasing at each iteration.

The *reverse*′ function of Figure 20 implements a naive algorithm for reversing the elements of a list.

$$\begin{array}{lcl} reverse'\,[\,] & = & [\,] \\ reverse'\,(x : xs) & = & append\ (reverse'\,xs)\ [x] \end{array}$$

This algorithm has quadratic heap space complexity, in contrast to the *revAux* and *reverse* functions, of linear heap space complexity:

$$\begin{array}{lcl} revAux\ [\,]\ ys & = & ys \\ revAux\ (x : xs)\ ys & = & revAux\ xs\ (x : ys) \\ reverse\ xs & = & revAux\ xs\ [\,] \end{array}$$

These differences become apparent in the results shown in Figure 20. □

$$
\begin{array}{l}
min :: Int \to Int \to Int \\
min\ x\ y \mid x \leq y = x \\
\qquad\quad \mid x > y = y \\
\\
minimumAc :: Int \to [Int] \to Int \\
minimumAc\ ac\ [\,] = ac \\
minimumAc\ ac\ (x : xs) = minimumAc\ (ac\ `min`\ x)\ xs \\
\\
removeAc :: Int \to [Int] \to [Int] \to [Int] \\
removeAc\ x\ [\,]\ ac = ac \\
removeAc\ x\ (y : ys)\ ac \mid x == y = revAux\ ys\ ac \\
\qquad\qquad\qquad\qquad\quad \mid x/ = y = removeAc\ x\ ys\ (y : ac) \\
\\
selectSortAc :: [Int] \to [Int] \to [Int] \\
selectSortAc\ [\,]\ ac = ac \\
selectSortAc\ xs\ ac = selectOrdAc\ (removeAc\ y\ xs\ [\,])\ (y : ac) \\
\qquad \mathbf{where}\ y = minimumAc\ (-\infty)\ xs
\end{array}
$$

$$
\begin{array}{rcl}
\sigma_{min} & \approx & 5 \\
\sigma_{minimumAc} & \approx & 11 \\
\sigma_{removeAc} & \approx & 9 \\
\sigma_{selectSortAc} & \approx & 17
\end{array}
$$

Figure 21: Tail-recursive selection sort algorithm

In *Safe*, tail-recursive functions are executed in constant stack space without the need of any compiler optimization [15]. Our algorithm *computeSigma* is aware of this, since it can be proven that for every tail-recursive function definition $f\ \overline{x}\ @\ \overline{r} = e_f \in \mathbf{FD}$ the result of $SD_f\ e_f\ (|\overline{x}| + |\overline{r}|)$ is always zero. As a consequence, we get $\sigma_0 = \sigma$, where $\sigma$ is defined as in *computeSigma*. If the latter does not depend on the input sizes, neither does $\sigma_0$.

**Example 13 (Tail recursive selection sort).** Let us consider the implementation of the *selection sort* algorithm shown in Figure 21, where every function builds its result in an accumulator parameter. As a consequence, all the functions are tail recursive, and their stack costs are constant (i.e. do not depend on the size of the input). Besides this, the upper bounds on the number of stack words inferred by *computeSigma* are also constant (see Figure 21). □

## 5. Inference in presence of explicit destruction and polymorphic recursion

Our inference algorithms yield correct upper bounds assuming the absence of region-polymorphic recursion. Besides this, *Safe* provides a **case**!

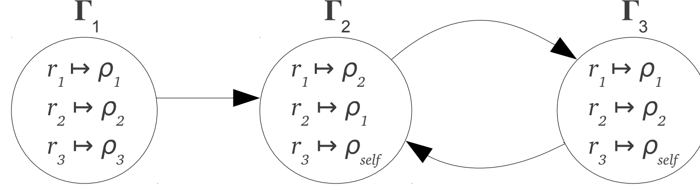
construct which, in addition to the pattern matching, destroys the cell pointed to by its discriminant. This kind of destruction is controlled by the programmer. In this section we briefly sketch how to improve these algorithms in order to deal with these language facilities. The implementation of these techniques is subject of future work.

If we consider region-polymorphic recursive definitions, the abstract interpretation described in Section 2.3.2 would require no changes, since its proof of correctness does not distinguish between recursive and non-recursive function applications. Polymorphic recursion does affect the computation of the initial $\Delta_0$ and $\mu_0$ explained in Section 2.4. Assume a function definition with $m$ region parameters, and that the type of the $i$-th region parameter is $\rho_i$, for each $i \in \{1..m\}$. The *computeDelta* algorithm assumes that this mapping between region parameters and RTVs remains constant through the subsequent recursive calls, but this is not true in the case of polymorphic recursive definitions: the $i$-th region parameter may be mapped to a different $\rho_j$ $(j \neq i)$ in some recursive calls, or it could be mapped to $\rho_{self}$. As a consequence, we have a finite number $\Gamma_1, \ldots, \Gamma_n$ of typing environments typing the subsequent recursive calls. For every recursive call, the correspondence between region variables and RTVs is given by one of these environments. Assume we are able to find some $nr_{(i)}$ $(i \in \{1..n\})$ such that $nr \sqsubseteq nr_{(1)} + nr_{(2)} + \cdots + nr_{(n)}$ and each $nr_{(i)}$ bounds the number of recursive calls associated with the environment $\Gamma_i$. Then we would be able to consider each $nr_{(i)}$ separately, multiplied by the charges done by the recursive subsequences under the environment $\Gamma_i$, in the style of the following example:

**Example 14.** Given the following region-annotated definition:

$$
\begin{array}{l}
f :: \mathit{Int} \to \rho_1 \to \rho_2 \to \rho_3 \to ([\mathit{Int}]@\rho_1, [\mathit{Int}]@\rho_2)@\rho_3 \\
f\ 0\ @\ r_1\ r_2\ r_3 = ([\,]@r_1, [\,]@r_2)@r_3 \\
f\ n\ @\ r_1\ r_2\ r_3 = (n : xs, n : ys)@r_3 \\
\qquad\qquad \textbf{where}\ (xs, ys) = f\ (n-1)\ @\ r_2\ r_1\ \mathit{self}
\end{array}
$$

Assume $\Gamma = [r_1 : \rho_1, r_2 : \rho_2, r_3 : \rho_3]$. In the first recursive call, the type of the first parameter becomes $\rho_2$ and the type of the second one becomes $\rho_1$, whereas no region is mapped to $\rho_3$. In the second recursive call, the type of the first parameter is $\rho_1$ again, and the type of the second parameter is $\rho_2$. In the third recursive call we have the same mapping as in the first one. These changes in the mappings between parameter positions and RTVs can be depicted as follows:




If $nr = \lambda n.n$ is an upper bound to the number of recursive calls, one of these calls is done with the typing environment $\Gamma_1$ above, at most $\lceil \frac{n-1}{2} \rceil$ of these calls correspond to $\Gamma_2$, and at most $\lceil \frac{n-1}{2} \rceil$ are done with $\Gamma_3$. Therefore, we have three different functions for modeling the number of recursive calls: $nr_{(1)} = \lambda n.1$, $nr_{(2)} = \lambda n. \lceil \frac{n-1}{2} \rceil$, and $nr_{(3)} = \lambda n. \lceil \frac{n-1}{2} \rceil$ By applying the abstract interpretation rules with the recursive and base part of the expression assuming each $\Gamma_i$, we obtain the following abstract heaps:

$$
\begin{array}{ll}
\Delta^r_{(1)} = [\rho_1 \mapsto 1, \rho_2 \mapsto 1, \rho_3 \mapsto 1] & \Delta^b_{(1)} = [\rho_1 \mapsto 1, \rho_2 \mapsto 1, \rho_3 \mapsto 1] \\
\Delta^r_{(2)} = [\rho_1 \mapsto 1, \rho_2 \mapsto 1] & \Delta^b_{(2)} = [\rho_1 \mapsto 1, \rho_2 \mapsto 1] \\
\Delta^r_{(3)} = [\rho_1 \mapsto 1, \rho_2 \mapsto 1] & \Delta^b_{(3)} = [\rho_1 \mapsto 1, \rho_2 \mapsto 1]
\end{array}
$$

Hence we get:

$$
\begin{array}{lcl}
\Delta_0 & = & nr_{(1)}\ n * \Delta^r_{(1)}\ n\ \rho_1 + nr_{(2)}\ n * \Delta^r_{(2)}\ n\ \rho_2 + nr_{(3)}\ n * \Delta_{(3)}\ n\ \rho_1 \\
 & & + nb\ n * (\Delta^b_{(1)}\ n\ \rho_1 \sqcup \Delta^b_{(2)}\ n\ \rho_2 \sqcup \Delta^b_{(3)}\ n\ \rho_1) \\
 & = & 1 + \lceil \frac{n-1}{2} \rceil + \lceil \frac{n-1}{2} \rceil + 1 \\
\Delta_0 & = & 1 + \lceil \frac{n-1}{2} \rceil + \lceil \frac{n-1}{2} \rceil + 1 \\
\Delta_0 & = & 1
\end{array}
$$

□

Regarding our destructive **case**!, the presented abstract interpretation needs to know which variable is affected by a **case**!. Assume we adapt our flattening transformation, so it generates sequences of the form $[G \to be_1, \ldots, be_n \mid D]$, where $D$ is a set containing the variables occurring in the discriminant of a destructive pattern matching. The $[\![\cdot]\!]_\Delta$ interpretation of sequences would be redefined as follows:

$$
[\![ [G \to \overline{be} \mid D] ]\!]_\Delta\ \Sigma\ \Gamma = \left[G \to \bigsqcup \left\{ [\,]_f, \textstyle\sum_{i=1}^{n} ([\![be_i]\!]_\Delta\ \Sigma\ \Gamma) + \sum_{x \in D} [R(x) \mapsto -1] \right\}\right]
$$

where $R(x)$ denotes the outermost RTV of the type of $x$. This information can be obtained from the typing environment. The least upper bound with

the empty abstract heap is done to avoid overall negative charges. With this new definition it can be proven that the $[\![\cdot]\!]_\Delta$ interpretation is correct, and yields a function in $\mathbb{D}$. However, the *computeDelta* algorithm is more problematic: it could return an initial bound $\Delta_0$ not belonging to $\mathbb{D}$, since the monotonicity property might not hold. However, in those examples where $\Delta_b$ and $\Delta_r$ (being these as defined in *computeDelta*) belong to $\mathbb{D}$, so does the resulting $\Delta_0$. The same applies to $\mu_0$ and $\sigma_0$: if their respective components belong to $\mathbb{F}$, so do the results $\mu_0$ and $\sigma_0$.

**Example 15.** Assume a destructive variant of the *revAux* function of Example 12.

$$
\begin{array}{l}
\mathit{revAuxD} :: [\alpha]@\rho_1 \to [\alpha]@\rho_2 \to \rho_2 \to [\alpha]@\rho_2 \\
\mathit{revAuxD}\ \mathit{xs}\ \mathit{ys}\ @\ r = \textbf{case!}\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad [\,] \to \mathit{ys} \\
\qquad\qquad\qquad\qquad\quad (x : \mathit{xx}) \to \textbf{let}\ x_1 = (x : \mathit{ys})@r\ \textbf{in}\ \mathit{revAuxD}\ \mathit{xx}\ x_1\ @\ r
\end{array}
$$

We get the following sequences:

$$
\begin{array}{rcl}
\mathit{seq}_1 & = & [\mathit{xs} \geq 1 \to \mathit{ys} \mid \mathit{xs}] \\
\mathit{seq}_2 & = & [\mathit{xs} \geq 2 \to (x : \mathit{ys})@r, \mathit{revAuxD}\ \mathit{xx}\ x_1 \mid \mathit{xs}]
\end{array}
$$

The first one is a base sequence, and the second one is recursive. We get the following abstract heaps by assuming $\Sigma(\mathit{revAuxD}) = ([\,]_f, 0, 0)$.

$$
\lfloor \Delta_b \rfloor = [\mathit{xs} \geq 1 \to [\rho_2 \mapsto 0]] \qquad \lfloor \Delta_r \rfloor = [\mathit{xs} \geq 2 \to [\rho_2 \mapsto 0]]
$$

Both of these heaps belong to $\mathbb{F}$. Since $nr = xs - 1$ and $nb = 1$, the *computeDelta* algorithm yields $\Delta_0 = [\mathit{xs} \geq 1 \to [\rho_2 \mapsto 0]]$. That is, *revAuxD* needs no additional heap space in order to be executed. □

## 6. Conclusions and future work

In this paper, we have showed that the memory bounds computed by the algorithms presented in [16] belong to the reductive subdomain of the abstract interpretation function, provided certain reasonable conditions are met. This property allows us to iterate the interpretation in order to obtain tighter, and still correct, bounds. Our approach then consists of finding some initial correct approximations, and repeatedly applying an abstract interpretation algorithm in order to increase their accuracy. The strengths of the whole approach can be summarized as follows:

1. It scales well to large programs, as bounds can be separately inferred for each *Safe* function.
2. It supports arbitrary algebraic data types, provided they do not present mutual recursion.
3. We get upper bounds for the maximum amount of *live* memory, as the inference algorithms take into account the deallocation of dead regions made at function termination.
4. It can accommodate several complexity classes, provided these are monotone with respect to the input sizes.
5. It is, to our knowledge, the first approach in which the upper bounds can be improved just by iterating the inference algorithm.

The latter point of the above list should be interpreted cautiously, since a larger amount of iterations implies more accuracy, but also more complexity in the resulting expressions (at least with our current implementation). Notice, however, that we still can get simple *asymptotic* bounds. In those cases where the programmer is only interested in asymptotic bounds, these can be given as input to the next iteration, and the result is another correct asymptotic bound, much simpler to compute than if the iteration were done with a non-asymptotic initial bound.

Notice also that, in some cases, it is possible to get an infinite, strictly decreasing sequence of upper bounds without reaching a fixed point. It is an interesting subject of future work to study how to automatically determine the limit of such sequences.

From the examples presented in Section 3.3, it can be seen that at present two approaches for using the system are possible:

- A completely automated way in which initial upper bounds are automatically inferred, then iterated a predetermined (small) number of times, and then printed. This mode is advisable when quick results are more important than precise ones, particularly if we just look for asymptotic costs.

- A semi-automated way in which human intervention is admissible, and more precise bounds are seeked for. By looking at the results given by the system, a human can compute the limits to which the sequences of bounds converge or even the fixpoints, and then introduce these bounds as input for improving the inference of subsequent functions.

A weak point that still requires more work is the restriction we have imposed to our cost and size functions: they must be non-negative and monotone. That is why destructive pattern matching has been omitted from our analysis in a first phase. Polymorphic recursion does not fit either in our inference strategy for $\Delta_f$, since it assumes that the regions used by a recursive function $f$ do not change from the external call to the internal ones. We have explained in Section 5 how our basic inference algorithms could be extended to cope with these features. Nevertheless, the ideas presented there have thus far neither still been implemented nor proved correct.

## References


[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, Closed-form upper bounds in static cost analysis, Journal of Automated Reasoning 46 (2011) 161–203.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost analysis of Java bytecode, in: Proceedings of the 16th European Symposium on Programming, ESOP'07, LNCS 4421, Springer, 2007, pp. 157–172.

[3] E. Albert, S. Genaim, M. Gómez-Zamalloa, Parametric inference of memory requirements for garbage collected languages, in: Proceedings of the 2010 international symposium on Memory management, ISMM'10, ACM Press, 2010, pp. 121–130.

[4] E. Albert, S. Genaim, A.N. Masud, More precise yet widely applicable cost analysis, in: Proceedings of the 12th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'11, LNCS 6538, Springer, 2011, pp. 38–53.

[5] J. Hoffmann, K. Aehlig, M. Hofmann, Multivariate amortized resource analysis, in: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL'11, ACM, 2011, pp. 357–370.

[6] J. Hoffmann, M. Hofmann, Amortized resource analysis with polynomial potential, in: Proceedings of the 19th European Symposium on Programming, ESOP'10, LNCS 6012, Springer, 2010, pp. 287–306.

[7] M. Hofmann, S. Jost, Static prediction of heap space usage for first-order functional programs, in: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2003, pp. 185–197.

[8] J. Hughes, L. Pareto, Recursion and dynamic data-structures in bounded space: towards embedded ML programming, in: Proceedings of the 4th ACM SIGPLAN international conference on Functional programming, ICFP'99, ACM, 1999, pp. 70–81.

[9] J. Hughes, L. Pareto, A. Sabry, Proving the correctness of reactive systems using sized types, in: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'96, ACM Press, 1996, pp. 410–423.

[10] S. Jost, Automated Amortised Analysis, Ph.D. thesis, Fakultät für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München, 2010.

[11] S. Jost, K. Hammond, H.W. Loidl, M. Hofmann, Static determination of quantitative resource usage for higher-order programs, in: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'10, ACM, 2010, pp. 223–236.

[12] M. Montenegro, R. Peña, C. Segura, A type system for safe memory management and its proof of correctness, in: Proceedings of the 10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, PPDP'08, ACM Press, 2008, pp. 152–162.

[13] M. Montenegro, R. Peña, C. Segura, A space consumption analysis by abstract interpretation, in: Proceedings of the 1st International Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA'09, LNCS 6324, Springer, 2010, pp. 34–50.

[14] M. Montenegro, R. Peña, C. Segura, A simple region inference algorithm for a first-order functional language, in: Proceedings of the 18th International Workshop on Functional and (Constraint) Logic Programming, WFLP'09, LNCS 5979, Springer, 2009, pp. 145–161.

[15] M. Montenegro, R. Peña, C. Segura, A resource semantics and abstract machine for Safe: A functional language with regions and explicit deallocation, Information and Computation 235 (2014) 3–35.

[16] M. Montenegro, R. Peña, C. Segura, Space Consumption Analysis by Abstract Interpretation, Technical Report, Dpto. de Sistemas Informáticos y Computación. Universidad Complutense de Madrid, 2014. TR-2-14, pages 1–104.

[17] M. Montenegro, R. Peña, C. Segura, Space Consumption Analysis by Abstract Interpretation: Reductivity Properties . Detailed Proofs, 2014. Pages 1–12.

[18] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer, 1999.

[19] L. Pareto, Sized types, 1998. Licentiate thesis, Chalmers University of Technology.

[20] H. Simões, P. Vasconcelos, M. Florido, S. Jost, K. Hammond, Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs, in: International Conference on Functional Programming (ICFP'12), ACM SIGPLAN, 2012, pp. 165–176.

[21] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pacific Journal of Mathematics 5 (1955) 285–309.