Space Consumption Analysis by Abstract Interpretation
Inference of Recursive Functions[☆]

Manuel Montenegro, Ricardo Peña, Clara Segura

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Abstract

We present 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. This paper begins by presenting *Safe* features by means of intuitive examples, and then defines its formal semantics, including the memory consumption of particular program executions. It continues by giving the abstract interpretation rules for non-recursive function definitions, and then how the memory consumption of recursive ones is approximated.

An interesting property of our analysis is that, under certain reasonable conditions, the inferred bounds are *reductive*, which 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. However, and due to lack of space, reductivity is not presented in this paper. The complete development can however be found in a technical report available at the authors' site.

The paper includes a related work discusion, and small examples. Bigger case studies are presented in the fore-mentioned technical report.

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

[☆]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)

1. Introduction

Among the set of desirable properties of a program, the most decisive ones are those related with its correctness, which ensures that a program does what the programmer expects it to do. This class of properties is commonly known as *functional properties*. Besides these, there are some other desirable properties that are relevant to the safety of software systems. These are called *non-functional properties*. An example is the fact that a program performs its task in a given amount of time, or that its memory needs do not exceed a given limit. These two examples become part of a broader research field, whose name is *resource analysis*. In this framework, a program is conceived as a resource consumer (resource may be understood as time, memory, energy, etc.) and the aim is to compute an upper bound to the resources being consumed by every possible execution of the program.

In this work, we are particularly interested in the analysis of *memory bounds*. Memory consumption is specially relevant to several scenarios: for instance, when programming embedded devices, it is necessary to make sure that the programs running in these devices do not stop working because they try to use more memory than it is available. It is also useful to know in advance how much memory will be needed by the program, in order to reduce hardware and energy costs. Although relatively new, the field of resource analysis has gained considerable attention in the last years, mainly due to the application of mathematical techniques (such as linear programming, or recurrence solving) to programming languages. In particular, the inference of memory bounds is a very complex task that involves several auxiliary analyses, each one a challenge by itself.

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 [24] 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 [25, 37].

The first fully automatic way to infer closed-form memory bounds is due to Hofmann and Jost [21]. Their analysis, based on a type system with resource annotations, can infer linear heap memory bounds on first-order eager functional programs with explicit deallocation. These techniques have been applied to subsets of imperative languages, such as Java [22] and C [18]. The annotated type system also serves as a basis for a stack consumption analysis due to Campbell [8, 9]. Hofmann and Jost's approach is extended in [27, 26] to higher-order programs. The latter work provides a general framework that can accommodate different notions of cost. More recently, Hoffmann and Hofmann have extended [21] to polynomial memory bounds [20, 19], and Simões et al [43] have extended it in a different direction: they provide a system computing the memory cost of functional programs with lazy evaluation.

The classical approach to resource analysis, due to Wegbreit [47], involves the generation of a recurrence relation from the program being analysed, and, in a second phase, the computation of a closed-form expression (without recursion) equivalent to that recurrence relation. Vasconcelos and Hammond pursue this approach in [46], which is fully automatic in the generation of recurrence equations, but requires the use of an external solver for obtaining a closed form. The COSTA system [3] follows a similar approach, but it provides its own recurrence relation solver, PUBS [2], 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. Since memory management in Java is based on garbage collection, their approach to memory consumption is parametric on the behaviour of the garbage collector [5]. The bounds computed by this system go beyond linear expressions; it can compute polynomial, logarithmic, and exponential bounds.

This paper describes the automatic analysis of memory bounds for a first-order functional language called *Safe*. This language 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. It was introduced for investigating the suitability of functional languages for programming small devices and embedded systems with strict memory requirements.

The absence of a notion of state makes reasoning about the functional properties of a program easier. Due to this lack of state, functional languages are in general better suited to several static analyses. However, the inference of memory bounds requires special attention. In most functional languages memory management is delegated to the runtime system, which allocates memory as it is needed by the program, provided there is enough space available. A *garbage collector* is in charge of determining, at runtime, which parts of the memory are no longer needed, and can be safely disposed of. The main advantage of this approach is that programmers do not bother about low-level details on memory management, but there are also some drawbacks. On the one hand, the time delay introduced by garbage collection may prevent a program from providing an answer in a required reaction time, which may be unacceptable in the context of real-time systems. There has been some successful work on real-time garbage collectors. For instance, in [42] the author guarantees a worst-case execution time by calling the memory recovery operations within the critical threads, but the price to be paid appears to be having a rather complex system. On the other hand, garbage collection makes it difficult to predict at compile time the lifetimes of data structures, specially in those cases where the runtime system does not specify under which conditions a garbage collection takes place.

In order to compute memory bounds for *Safe*, we have decided to dispense with the garbage collector and to have a heap structured as a region stack in which regions are allocated and deallocated in constant time. Given this memory model, we use abstract interpretation-based techniques [14] for inferring non-linear, monotonic, closed-form expressions bounding the heap and stack memory costs of a program.

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. The bounds given by this memory consumption analysis are considered correct if they are equal to or greater than the actual worst-case runtime consumptions of the program being analysed.

The problem of inferring memory bounds is closely related to the inference of *size relations* between data structures. The seminal work on type-based sizes is due to Hughes, Pareto, and Sabry [25], which is restricted to type checking. The inference problem is addressed by Chin and Khoo [11]. Another approach by Benoy and King [7] uses abstract interpretation-based techniques on the domain of convex polyhedra. Abstract interpretation is also applied in [44] to the approximation of the height[1] of data structures. All these techniques are restricted to linear size relations. The work of Shkaravska, van Eekelen and van Kesteren [40] is able to infer polynomial size

[1]The height of a data structure is the longest chain of pointers that can be followed from the initial one.

relations. It provides a type system, in which checking is decidable under certain syntactic conditions, and type inference is performed by a combination of testing and polynomial interpolation-based techniques.

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. Therefore, we shall be modest in this work, and focus on the inference of heap and stack memory bounds by assuming that the size and call-tree information is given externally. As reported above, the PUBS system and the work by Shkaravska et al., have provided respectively partial solutions to recurrence solving and size analysis. We have contributed to the first problem in [28] and [38], being also the latter a contribution to the second one regarding linear size relations.

This paper is an extended and improved version of [30]. A major difference w.r.t. that paper is that here we present a method for flattening an expression into sequences of basic expressions. This makes the algorithm simpler when considering the base and recursive cases of a function definition, and also makes the analysis more precise. We can summarize the original contributions of [30] and this extended version as follows:

- We infer space bounds for a functional language with lexically scoped regions. There has been previous work on region inference and on space analysis for functional languages, but this appears to be the first work combining both.

- We use abstract interpretation directly on the infinite domain of multivariate monotonic functions. Other works either use special type systems or use abstract interpretation in polyhedra domains for some subproblems such as inferring linear size relations.

- Our bounds go beyond multivariate polynomials. The shape of our symbolic bounds partially depends on the shape of the symbolic functions we receive as a result of the size and call-tree depth analyses.

- 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 the bound as the input of a new interpretation, we get a new bound which is not only correct but also tighter. The advantage

$$
\begin{array}{l}
\mathit{insert}\ y\ [\,] = [y] \\
\mathit{insert}\ y\ (x : xx) \\
\qquad\quad \mid y \leq x = y : x : xx \\
\qquad\quad \mid \text{otherwise} = x : \mathit{insert}\ y\ xx \\
\\
\mathit{insSort}\ [\,] = [\,] \\
\mathit{insSort}\ (x : xx) = \mathit{insert}\ x\ (\mathit{insSort}\ xx)
\end{array}
$$

Figure 1: Insertion sort algorithm in *Safe*

of having reductive bounds is that they allow us to obtain a possibly decreasing (at least a non-increasing) sequence of bounds as a result of iterating the analysis. For lack of space, this aspect is explained and proved correct in a separate paper. The complete development can be found in [34].

- We have formally proved the correctness of all the results contained in this paper. So the analysis is correct with respect to the memory cost model specified by the language semantics.
- 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 [35].

In order to give a flavour of the kind of results the reader will find in the rest of the paper, let us consider the well known *insertion sort* algorithm whose *Safe* text is given in Figure 1.

By calling xs to the size on the input list expressed in terms of the number of constructor applications it contains, our space inference algorithm gets as a first approximation of the heap consumption in terms of constructor cells, and of the stack consumption in terms of words, the following functions:

$$
\begin{array}{ll}
\mu_0 = xs^2 - 2xs + 3 & \forall xs \geq 5 \\
\sigma_0 = 14xs - 13 & \forall xs \geq 3
\end{array}
$$

By iterating the interpretation using the above functions as input, our algorithm gets:

$$
\begin{array}{ll}
\mu_1 = xs^2 - 3xs + 6 & \forall xs \geq 6 \\
\sigma_1 = 14xs - 21 & \forall xs \geq 4
\end{array}
$$

which are still correct but smaller bounds. It is instructive to compare these bounds with the exact bounds one can compute from the semantics (Section 2.3 explains how to do this exact computation) for the worst case of the algorithm:

$$\begin{array}{ll} \mu = \frac{1}{2}xs^2 + \frac{1}{2}xs & \forall xs \geq 1 \\ \sigma = 8xs - 12 & \forall xs \geq 4 \end{array}$$

In this example, in a first approximation our system is able to infer the same order of complexity as that of the exact bounds, but we lose some precision due to the over approximations made in several places of the abstract interpretation. This loss is typical in most of the examples.

Plan of the paper

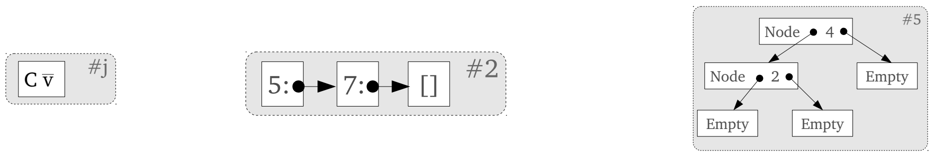
After this introduction, in Section 2 we present *Safe* by means of intuitive examples, ending in a formal semantics including the memory consumption of particular executions. Then, in Section 3 we give the abstract intepretation rules of *Safe* expressions, assuming that the memory consumption of the called functions is known and is kept in a global *function signature environment*. We prove these rules correct with respect to the language semantics. In Section 4 we show how the memory consumption signature of recursive functions is approximated, and prove the inference algorithms correct. Finally, Section 5 reviews related and future work, and concludes.

2. Syntax and resource-aware semantics of *Safe*

2.1. Language concepts: Safe by example

Safe is a first-order polymorphic functional language, whose syntax is similar to that of (first-order) Haskell or ML, but with some facilities to manage memory. Polymorphic data types are defined in the same way as in Haskell. Functions are defined as a set of equations with the same syntax as Haskell functions. The purpose of *Safe* is to serve as a research platform to prove the suitability of functional languages for programming embedded devices and safety critical systems. One of its aims is to analyse memory consumption and to provide formal certificates of the correctness of the inferred bounds. The certification aspect has been presented elsewhere [15].

Safe's memory model is based on *heap regions*. Regions are disjoint parts of the heap where data structures are built. A region can be created and disposed of in constant time.



(a) Cell (b) List (c) Binary tree

Figure 2: Graphical representation of cells and regions.

A *cell* is a piece of memory big enough to hold a data constructor with its parameters. In implementation terms, a cell contains the identifier of a data constructor, and a representation of the values to which this constructor is applied. These values can be either basic (integers or booleans), or pointers to other cells. With the term "big enough" we mean that a cell being disposed of the heap may be immediately reused by the runtime system. A naive implementation would define this size as the space taken by the biggest constructor (i.e. with the highest number of parameters). In a more efficient approach there would be a fixed number of cell sizes, all of them multiple of the smallest one. In any case, the aim is to reuse a cell in constant time.

We represent regions and cells as in Figure 2. A cell is depicted as a white square which contains the constructor and the arguments to which it is applied. Pointers are represented via arrows between cells, whereas basic values are shown in the cell itself. Shaded rectangles correspond to regions, which are labelled with a number identifying them (see Figure 2a). As an example, Figure 2b shows a list of integers, in which the constructor (:) is shown in infix form.

Cells are combined in order to build *data structures.* A data structure (DS in the following) is the set of cells that results from taking a particular cell (the *root*) and following the transitive closure of the relation $C_1 \to C_2$, which denotes that C_1 and C_2 are cells *of the same type,* and there is a pointer in C_1 to C_2. An important thing to note is that we only consider as part of a DS the set of cells with the same type as the root cell. For instance, if we have a list of lists (type $[[\alpha]]$), the cells that make up the recursive spine of the outer list constitute a DS, to which the inner lists do not belong, even when there are pointers from the outer list to them. Each one of the inner lists constitute a separate DS on its own.

During the design of the language several decisions (axioms) were taken:

1. *A DS completely resides in a single region. The rationale behind this is to prevent creating dangling pointers when a region is deallocated.*
2. *A DS can be part of another DS, and two DS may share a third DS.*
3. *Basic values (integers and booleans) occurring in the heap do not belong to any region by themselves. They are contained within cells.*
4. *Allocation of regions takes place at function calls. Deallocation of regions takes place when a function call finishes.*

Decision (1) poses a constraint to the data constructors: the recursive children of a cell (i.e. those with the same type) must belong to the region of the father. As an example of (2), consider the binary tree of Figure 2c. The left and the right subtrees of the root are separate DSs, which belong to the whole binary tree, which is another DS.

Regarding (4), a distinctive aspect of *Safe* is the way in which regions are created and destroyed: new regions are created as functions are called, so there exists a correspondence between the function call stack and regions, which are also created and disposed of in a stack-like fashion. Since function calls have nested lifetimes (for instance, if f calls to a function g, the execution of the latter begins after the execution of f has started, and it finishes before the execution f has finished), regions also have nested lifetimes.

The region associated to a given function call f is called its *working region*. The function may create DSs in this region, provided these are not accessed outside the function's context, since they will be destroyed when the function finishes. A function may also access the working regions of the function calls situated below it in the call stack. These regions must be passed as parameters by the functions calling f. Each region existing at a given execution point is uniquely identified by a natural number ranging from 0 (which identifies the bottommost region in the stack) to the number k of active regions minus one (which identifies the topmost one).

An important point is the fact that regions are not handled directly by *Safe*'s programmers. The compiler determines which DSs will be created in the working region and which regions should be passed as parameters between functions [32]. However, in order to get an idea on how regions are inferred, we will consider a syntactically-extended version of *Safe*, which we call *Safe with regions*. In this version regions become apparent. The main syntactical additions of *Safe* with regions include the following:

- A function definition may have additional region parameters $r_1 \ldots r_m$ separated by a @ from the rest of formal parameters. As an example, we may have the following function definition:

$$f\ x_1\ x_2\ x_3\ @\ r_1\ r_2 = \ldots$$

These extra parameters will contain, at runtime, the identifiers of the regions in which f will build its output.

- The working region is referred to by the identifier *self*.

- When calling a function, the actual region arguments are also separated from the rest of the arguments by the @ symbol. For example, $f\ 4\ x\ z\ @\ \mathit{self}\ r_1$, where r_1 is a region variable in scope.

- Each constructor expression is attached a region variable which contains, at runtime, the identifier of the region where the resulting cell will be built. For example, $[\,]\ @\ r_2$, or $(4\ :\ [\,]\ @\ \mathit{self}\ )\ @\ \mathit{self}$. In the latter example, the outermost *self* annotates the application of the list constructor (:).

Example 1. Consider a function *append* for concatenating two lists. The following is *Safe* code, as written by the programmer:

$$\begin{array}{llll} \mathit{append} & [\,] & \mathit{ys} & = \mathit{ys} \\ \mathit{append} & (x : \mathit{xs}) & \mathit{ys} & = x : \mathit{append}\ \mathit{xs}\ \mathit{ys} \end{array}$$

This function is annotated by the compiler as follows:

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

There is a new region parameter r, which is used to build the resulting list, and is passed to the subsequent recursive calls. □

The working region *self* of a function is used to build temporary DSs which are not part of the result. An example of a function with this kind of behaviour is *treesort*.

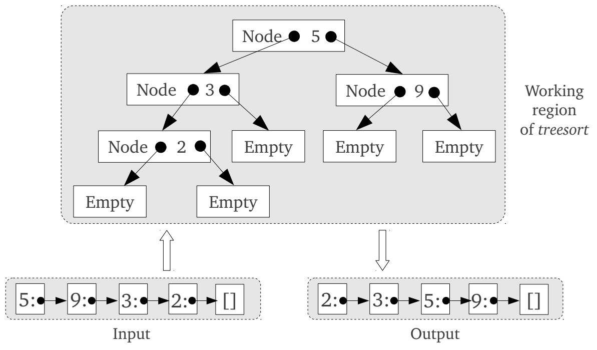


Figure 3: DSs involved in the *treesort* function.

Example 2. A tree sort algorithm builds a binary search tree from the input list to be sorted. Then it does an inorder traversal of the tree, so that the elements come out in sorted order. Assume the following implementation,

$$treesort\ xs = inorder\ (mkTree\ xs)$$

where *mkTree* builds a binary search tree from the list given as parameter, and *inorder* performs an inorder traversal of a binary search tree by adding the visited elements to a list that is returned as result. Now we show the *Safe* code with regions:

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

Both functions *inorder* and *mkTree* receive a region parameter specifying where to build the resulting list (resp. tree). The *mkTree* function is given the *self* identifier, so the tree will be built in the working region of *treesort*. The *inorder* function receives the parameter given to *treesort*, which is the output region in which the sorted list will be built (see Figure 3). When *treesort* finishes, its working region will disappear from the heap, together with the temporary tree. □

Safe provides a built-in facility for copying data structures: the @ notation. The expression *ys*@ returns a copy of the DS pointed to by *ys*. The copy of the data structure will be located in a (possibly) different region, if this does not contradict axiom (1). The copy facility is useful when the programmer does not want to build a DS upon already existing ones.

Example 3. The *append* function of Example 1 forces the resulting list to be located in the same region as the list passed as second parameter. This is because the result is linked to this parameter, so that the latter becomes part of the former, and, by axiom (1), they must live in the same region. Let us consider the following variant in which the result is built upon a *copy* of the list passed as second parameter:

$$\begin{array}{llcl} appendC & [\,] & ys & = & ys@ \\ appendC & (x : xs) & ys & = & x : appendC\ xs\ ys \end{array}$$

The compiler annotates every copy expression with the region variable in which the copy will be returned. In the case of *appendC* function, it produces the following code with regions:

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

The copy of *ys* is created in the output region r, which may now be different from the region of the second parameter *ys*. □

2.2. Full-Safe vs Core-Safe

The functions presented previously were written in *Full-Safe*, which is the language in which the programmer writes his programs. However, *Full-Safe* results cumbersome when designing program analyses, since the number of language syntactic constructs to consider becomes overwhelming. For this reason, we have a simplified variant of *Full-Safe* (which is called *Core-Safe*), with a fewer number of syntactic expressions. This approach is similar to that of the translation of Haskell programs into a Core language, as done in the GHC compiler [23]. The details of the translation phase from *Full-Safe* to *Core-Safe* are beyond the scope of this work (see [12] for details), but this process follows these general guidelines: (1) each function is represented by a single equation; (2) pattern matching is translated into **case** expressions; (3) region variables are made explicit in *Core-Safe*; and (4) only atomic expressions (constants and variables) are allowed in function and constructor applications. Non-atomic expressions occurring inside function and constructor arguments must be introduced via **let** bindings, in the style of A-normal form [17]. For instance, $f\ (2 + 4)$ is transformed into **let** $z = 2 + 4$ **in** $f\ z$.

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

Figure 4: *Core-Safe* language definition.

Example 4. The translation phase applied to the *append* function defined previously yields the following result:

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

□

Since the analysis described in this paper works at the *Core-Safe* level, this language is described in detail below. However, and for the sake of clarity, we will use *Full-Safe* for most medium- and large-sized examples. We shall even use region-annotated *Full-Safe* code, when regions are relevant.

In Figure 4 we show the syntax of *Core-Safe* programs and expressions. We use the abbreviation $\overline{s}$ to denote the item sequence $s_1, \ldots, s_n$. We will refer to each of such items as s_i or s_j. When the sequence length n is important, we will write $|\overline{s}| = n$.

A program *prog* is a sequence $\overline{data}$ of **data** declarations, followed by a sequence $\overline{def}$ of function definitions and a main expression e, whose result is the result of the program.

2.2.1. Data types declarations

In a **data** declaration α denotes a polymorphic type variable, ρ a polymorphic *region type variable* and t a type. A **data** declaration follows a syntax similar to that of Haskell, with the addition of the region type variables. When extending the Hindley-Milner type system to *Core-Safe*, one must assign a type to region variables. For this purpose we have defined a new category of types: *region type variables*. The type of a region variable is a region type variable (abbreviated as RTV in the following). These RTVs, which will be denoted by $\rho, \rho_1, \ldots$, act as ordinary polymorphic variables in Haskell. However, only region variables are allowed to have a RTV as its type. The set of RTVs is denoted as **RegType**.

Algebraic data types are annotated with RTVs, which always coincide with the types of the region variables used in the creation of the corresponding DS. For example, if r has type ρ, the expression $[\,]\ @\ r$ has type $[\alpha]@\rho$. We have these kind of annotations with the aim of stating a connection between data structures and region variables, and connections among different data structures. For example, if two variables have $[\alpha]@\rho$ and $[\beta]@\rho$ as their respective types, their corresponding lists must live in the same region at runtime. Moreover, if there exists another region variable r' with type ρ, every DS being constructed with this variable at runtime will also live in the same region as these two lists. It is important to have a way to determine these connections at compile time, because it allows us to know, in particular, whether a given data structure resides in the temporary region *self*, which is the only region variable of type ρ_{self}.

Algebraic data types can be associated with more than one RTV. For example, the following definition

$$\mathbf{data}\ \mathit{TBL}\ \alpha\ \beta\ @\ \rho_1\ \rho_2\ \rho_3 = \mathit{TBL}\ [(\alpha,\beta)@\rho_1]@\rho_2\ @\ \rho_3$$

defines a concrete implementation of the table abstract data type, represented as a list of $(\mathit{key}, \mathit{value})$ pairs. Data structures of this type may spread up to three different regions: one for the *TBL* constructor, another for the spine of the list containing the pairs, and another one for the pairs themselves. The last RTV of the list $\overline{\rho}$ is the type of the region where the data structures of this type are built. This RTV is called the *outermost region*. In our example, the outermost region of the TBL data type is ρ_3.

2.2.2. Functions and expressions

A function definition consists of a global name f, followed by a list of formal parameters $\overline{x}$ (which are variables), a list of formal region parameters $\overline{r}$ (which are region variables) and the function body e. The sets of function symbols, variables and region variables are respectively denoted by **Fun**, **Var** and **RegVar**.

We denote by **Exp** the set of *Core-Safe* expressions. Basic expressions **BExp** include: atomic expressions (literals or variables), copy expressions, function and constructor applications, and a special kind of function applications that we consider to be built-in: basic operator applications. The set of basic operators $\oplus$ is left unspecified. We only demand that applications of these operators require no additional heap space and only two stack words for the arguments.

We assume the existence of a set **Cons** of data constructor names, and that, for every constructor C, the set of its recursive positions (denoted by $RecPos(C)$) is known at runtime. For instance,

$$\begin{array}{rr} RecPos([\,]) = \emptyset & \quad RecPos(:) = \{2\} \\ RecPos(Empty) = \emptyset & \quad RecPos(Node) = \{1,3\} \end{array}$$

We also assume that there is no mutual recursion between functions. This is done for the sake of simplicity. Everything can be adapted with relative ease in order to support mutual recursion.

The **let** construct allows having non-recursive, monomorphic intermediate declarations. Throughout this paper we use the terms *auxiliary* and *main* expression to refer to its component expressions e_1 and e_2, and we will usually write **let** $x_1 = e_1$ **in** e_2.

Pattern matching is supported via **case** expressions.

2.3. Resource-aware semantics

In Figure 5 we show the resource-aware big-step operational semantics of *Core-Safe* expressions.

2.3.1. Judgements

A judgement $E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$ means that expression e successfully reduces to a value v under a runtime environment E and a heap

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

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

$$\frac{}{E \vdash h, k, td, a_1 \oplus a_2 \Downarrow h, k, E(a_1) \oplus E(a_2), ([\,], 0, 2)}\ [\mathit{PrimOp}]$$

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

$$\frac{\begin{array}{c}(g\ \overline{y}\ @\ \overline{r}' = e_g) \in \mathbf{FD} \qquad |\overline{y}| = n \qquad |\overline{r}'| = l \\ [\overline{y} \mapsto E(\overline{a}), \overline{r}' \mapsto E(\overline{r}), \mathit{self} \mapsto k+1] \vdash h, k+1, n+l, e_g \Downarrow h', k+1, v, (\delta, m, s)\end{array}}{E \vdash h, k, td, g\ \overline{a}\ @\ \overline{r} \Downarrow h' \mid_k, k, v, (\delta \mid_k, m, \max\{n+l, s+n+l-td\})}\ [\mathit{App}]$$

$$\frac{E(r) = j \qquad j \leq k \qquad \mathit{fresh}_h(p)}{E \vdash h, k, td, C\ \overline{a}\ @\ r \Downarrow h \uplus [p \mapsto (j, C\ E(\overline{a}))], k, p, ([j \mapsto 1], 1, 1)}\ [\mathit{Cons}]$$

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

$$\frac{E(y) = p \quad h(p) = (j, C\ \overline{v}) \quad |\overline{v}| = n \quad E \uplus [\overline{x} \mapsto \overline{v}] \vdash h, k, td+n, e \Downarrow h', k, v', (\delta, m, s)}{E \vdash h, k, td, \mathbf{case}\ y\ \mathbf{of}\ \{\ldots; C\ \overline{x} \rightarrow e; \ldots\} \Downarrow h', k, v', (\delta, m, s+n)}\ [\mathit{Case}]$$

Figure 5: Resource-aware operational semantics of *Core-Safe* expressions.

h with $k+1$ regions (ranging from 0 to k) and that a final heap h' with the same number $k+1$ of regions[2] is produced as a side effect.

The triple (δ, m, s) is a *resource vector*. It represents the resource consumption of e, and can be conceived as a side effect of evaluating the expression. Core-Safe is translated to the code of an imperative machine called *Safe Virtual Machine* (SVM). The resource vector is formally derived [31, 33] from this translation and describes the memory needs of each syntactical construction of the language in terms of the SVM heap and stack. Components δ and m respectively represent the incremental and peak heap consumption, while s represents the peak stack consumption. The concrete values this resource vector may take are explained below.

We now explain in detail the other elements of the operational semantics. We use $v, v_1, \ldots$ metavariables to denote *values*, which are defined by the

[2]Actually, the latter k is redundant, as the final heap always has the same number of regions as the initial one. However, in this paper we shall make the k of the final configuration explicit.

following grammar:

$$\begin{array}{llll} \mathbf{Val} \ni v & ::= & p \in \mathbf{Loc} & \{ \text{ heap pointer } \} \\ & \mid & c \in \mathbf{Int} \cup \mathbf{Bool} & \{ \text{ literal: integer or boolean } \} \end{array}$$

A *heap* h is defined as a finite mapping from heap pointers to construction cells. *Heap pointers* specify memory locations. We assume the existence of a denumerable set of pointers **Loc** and use $p, p_1, q, \ldots$ to denote elements from this set. A *construction cell* w is an element of the form $(j, C\,\overline{v})$, where j is a natural number, $C \in \mathbf{Cons}$ a constructor symbol of arity n, and $\overline{v}$ is the n-list of values to which C is applied. The number j stands for the region of the heap in which the cell is located. With this heap model the region number may be considered as a property of a cell. This implies, on the one hand, that every cell belongs to a region and, on the other hand, that every cell belongs to a *single* region (in other words, regions are *disjoint*). For example, the following mapping h_0 models the heap shown in Figure 2b:

$$h_0 = \begin{bmatrix} p_1 & \mapsto & (2, 5 : p_2) \\ p_2 & \mapsto & (2, 7 : p_3) \\ p_3 & \mapsto & (2, [\,]) \end{bmatrix}$$

The notation $\mathit{fresh}_h(p)$ denotes that the pointer p is fresh in h, that is, it does not occur neither in its domain nor in their cells.

A *runtime environment* E (also called *value environment*) is a partial function mapping program variables x to values, and region variables r to actual regions (i.e. natural numbers) in the heap. We adopt the convention that, for every value environment E, if c is a literal, $E(c) = c$. Also, by $E(\overline{x})$ we denote the sequence $E(x_1), \ldots, E(x_n)$.

We assume that, during the evaluation of an expression, an environment **FD** of program function definitions is propagated through the $\Downarrow$ judgements. This environment maps function names to function definitions.

The semantics of a program $\mathit{prog} \equiv \overline{\mathit{data}}; \overline{\mathit{def}}; e$ is the result of evaluating its main expression e in an environment **FD** containing all the function declarations $\overline{\mathit{def}}$, under an empty heap with a single region 0 and a value environment which maps the *self* identifier to that region:

$$[\mathit{self} \mapsto 0] \vdash [\,], 0, e \Downarrow h', 0, v, (\delta, m, s) \tag{1}$$

2.3.2. Resource consumption

The resource vector (δ, m, s) captures the following information:

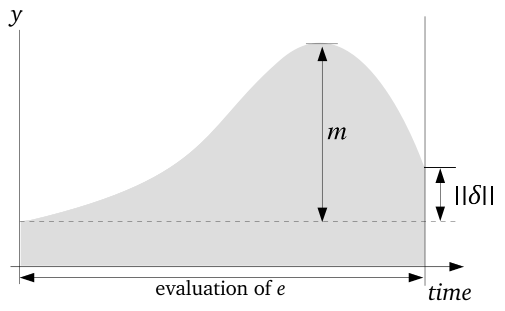


Figure 6: Intuitive meaning of δ and m components in the resource vector. The y coordinate represents the number of cells in the heap.

- The first component is a partial function $\delta : \mathbb{N} \to \mathbb{N}$ giving, for each region k, the difference between the number of cells after and before evaluating the expression. This difference can only be positive or zero. In an extended version of *Core-Safe* in which we allow explicit deallocation of cells [29], this difference could be also negative.

- The component m is a natural number describing the *minimum* number of fresh cells needed in a heap to successfully evaluate e, i.e. the *maximum* heap memory consumed by e. Some authors refers to this maximum value as the *peak* heap memory of e.

- The component s is a natural number whose meaning is analogous to that of the m component. The s component describes the minimum number of words needed in the stack for the evaluation of expression e. We could also refer to it as the *peak* stack memory of e.

Figure 6 gives an intuition on the meaning of the first two components. Assume the evaluation of an expression e. The figure represents the global amount of cells in memory as the evaluation of e proceeds. In this case, the evaluation of e reclaims memory until some time point, from which memory is disposed of. The m value represents the maximum amount of memory taken during the evaluation of e, whereas δ represents the difference of memory amount between the initial and final heaps. Notice, however, that the δ contains this difference *for every region* in the heap. What is represented in Figure 6 is the global balance $\|\delta\|$ of heap cells between the final and initial heaps, formally defined below. Also notice that both values m and δ are relative to the memory consumption level at the beginning of the evaluation of e (dashed line in Figure 6).

The domain of δ is the set $\{0..k\}$, where k is the number of regions in the heap to which the δ refers. The notation $[\,]$ stands for the function $[i \mapsto 0 \mid i \in \{0..k\}]$, where the value of k is assumed to be clear from the context. The notation $[i \mapsto n]$ abbreviates the function $[i \mapsto n] \uplus [j \mapsto 0 \mid j \in \{0..k\}\backslash\{i\}]$. The *total balance* of cells, denoted by $\|\delta\|$, is the sum of the balances obtained in each region:

$$\|\delta\| \overset{\text{def}}{=} \sum_{i \in \operatorname{dom} \delta} \delta(i) \tag{2}$$

The notation $\delta_1 + \delta_2$ represents the component-wise addition of δ_1 and δ_2, provided that they have the same domain.

Regarding the resource vector's third component s, if we represented the stack consumption in the style of Figure 6, the s component would take the role of the m component in the heap consumption. The final stack contains the same elements of the initial one and, in addition, the result of evaluating the expression at the top, so the difference between them is always 1.

Stack is mainly consumed in function application and pattern matching. When a function application is executed, first the actual parameters are stacked. Since the execution of the function being called occurs in a different context, the previous environment is partially discarded before the evaluation of the function body. This feature allows us having constant stack space for tail-recursive functions. In the semantics, we use a component td (named so after *top depth*) representing the number of stack words of the previous environment which can be discarded at this function application. It influences the stack consumption (see the *App* rule), since td words are removed from the stack before entering the function body.

2.3.3. Semantic rules

Now we explain in detail the semantic rules. Rules [*Lit*] and [*Var*] just say that literals and heap pointers are normal forms. Their evaluation does not consume heap, but it requires a stack word to push the result into. The evaluation of a primitive operator application requires two stack words, since the operands have to be pushed into the stack before computing the result.

Rule [*Copy*] copies the data structure pointed to by p and living in a region j' into a (possibly different) region j. The runtime system function *copy* follows the pointers in recursive positions of the structure starting at p and creates in region j a copy of all recursive cells. The normal form becomes

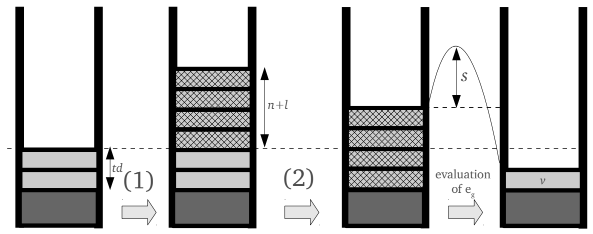


Figure 7: Stack consumption while evaluating a function application

a fresh pointer p' pointing to the copy. The pointers in non recursive positions of all the copied cells are kept identical in the new cells. This implies that both data structures (the original and the copy), may share some subparts.

The evaluation of a copy expression $[Copy]$ requires as many heap cells as the size of the recursive spine of the structure being copied. The *size* function, formally defined in Section 3, captures this amount of cells.

Rule $[App]$ shows when a new region is allocated. Notice that the function body is executed in a heap with $k+2$ regions (from 0 to $k+1$). The formal identifier *self* is bound to the newly created region $k+1$ so that the function body may create DSs in this region or pass this region as a parameter to other function calls. Before returning from the function, all cells created in region $k+1$ are deleted. This action could be a source of dangling pointers, but our region inference algorithm [32] decorates programs with regions in such a way that they never arise for this reason. By the notation $h\mid_k$ we denote the heap obtained by deleting from h those bindings living in regions greater than k:

$$h \mid_k \stackrel{\text{def}}{=} h \mid_{P(k,h)} \qquad \text{where } P(k,h) = \{p \in \text{dom}\, h \mid \mathit{region}(h(p)) \leq k\}$$

By $\delta|_k$ we mean a function like δ but restricted to the domain $\{0..k\}$.

The computation $\max\{n+l, s+n+l-td\}$ of fresh stack words reflects the actions taking place at function application, as depicted in Figure 7: (1) The function arguments are inserted at the top of the stack; (2) The td topmost words of the prior environment are discarded, while the $n+l$ arguments are slid down. Then, the function body is executed, needing s fresh words. Hence the above computation.

Rule $[Cons]$ generates a fresh location p pointing to the newly constructed cell in the corresponding region. It also requires a stack word to push p.

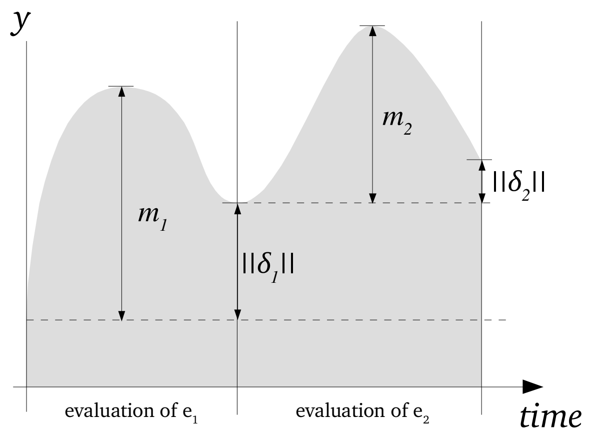


Figure 8: Heap consumption while evaluating a a **let** expression

Rule [*Let*] shows the eagerness of the language: first, the auxiliary expression e_1 is reduced to normal form and then the main expression e_2 is evaluated. In the latter evaluation the environment is extended by binding the program variable x_1 to the normal form to which e_1 is reduced.

Regarding memory consumption, when a **let** expression is evaluated, a continuation is stacked before evaluating e_1 so that execution can proceed later with e_2. Those continuations are counted in the stack consumption as two words. So, two words are stacked before evaluating e_1, and the evaluation leaves a 1-word value in the stack before evaluating e_2. That is why we obtain $\max\{2 + s_1, 1 + s_2\}$ as stack consumption. With regard to the heap consumption, the δ component is clearly additive, as depicted in Figure 8. The execution of the first expression leaves $\|\delta_1\|$ cells in the heap. This point is the reference level on which we measure the memory needs m_2 of the second expression. Hence, the peak memory needs of the whole sequence is given by $\max\{m_1, \|\delta_1\| + m_2\}$, as Figure 8 shows.

The [*Case*] rule is the usual one for an eager language. Besides the consumption of the executed alternative, the pattern matching requires n additional stack positions, being n the number of arguments of the corresponding constructor.

Example 5. In the following table we show the resource vector corresponding to the execution of some examples in this section:

$xs = [1, 2, 3]$ (4 cells) $\quad ys = [4, 5]$ (3 cells) $\quad zs = [5, 4, 3, 2, 1]$ (6 cells)

$t = Node\ (Node\ Empty\ 2\ Empty)\ 4\ (Node\ Empty\ 7\ Empty)$ (7 cells)

Expression	δ	m	s
$append\ xs\ ys\ @\ r$	$[E(r) \mapsto 3]$	3	23
$appendC\ xs\ ys\ @\ r$	$[E(r) \mapsto 6]$	6	24
$mkTree\ zs\ @\ r$	$[E(r) \mapsto 26]$	26	49
$inorder\ t\ @\ r$	$[E(r) \mapsto 4]$	4	19
$treesort\ zs\ @\ r$	$[E(r) \mapsto 6]$	32	53

Function *append* creates as many cells in the output region as the number of cons cells in the list passed as first parameter. In *appendC* we need three additional cells for copying the list passed as second parameter. Function *treesort* leaves in the output region as many cells as the input list. However, more cells are needed in order to build the intermediate tree. □

3. Analysis of nonrecursive definitions by abstract interpretation

Our space consumption analysis is expected to compute, given an expression e, upper bounds to each component of the resource vector (δ, m, s) resulting from its evaluation, which will be called (Δ, μ, σ). First we identify the elements of our *abstract domain* that can be used to express an upper bound to each component. After this, we explain how to infer such elements from the expressions of the program via an abstract interpretation function. Finally, we state some correctness results on the latter.

3.1. Abstract domain of function signatures

Assume a function definition of n data parameters and m region parameters:

$$\begin{array}{l} f :: t_1 \rightarrow \cdots \rightarrow t_n \rightarrow \rho_1 \rightarrow \cdots \rightarrow \rho_m \rightarrow t \\ f\ \overline{x}\ @\ \overline{r} = e_f \end{array} \tag{3}$$

The space consumption of f does not solely depend on the body of the function e_f, but also on the input parameters $\overline{x}$, which will be abstracted by their *sizes*.

Our first step is to define a suitable notion of *size* which reflects how easy or difficult is to process an input parameter. The memory costs of a *Safe* function are described as a function on this size.

Example 6. As an example, function *append* defined in Example 1 creates as many heap cells (all of them in the region argument r) as list elements has its first argument xs, because once the list xs is traversed the list ys becomes the tail of the result, so no nil constructor is built. It also consumes seven stack words per list constructor, both cons and nil, of xs. If we use the same name xs to represent its number of list constructors, the heap consumption is $xs - 1$ (only the cons) while the stack consumption is $7xs$. □

Following the idea shown in the example, the *size* function, when applied to a pointer, takes into account the cells of the *recursive spine* of the corresponding DS. For instance, if p points to a list of integers, the size of p is the number of integers plus one, not the sum of the integers themselves. In general, the size of a list with n elements is $n + 1$, since it is made of n cells with the (:) constructor plus an additional cell with the [ ] constructor. Analogously, the size of a binary search tree (as defined in Section 2.1) with n elements is always $2n+1$ (n cells with the *Node* constructor and $n+1$ cells with the *Empty* constructor).

Definition 1. Given a heap h and a value v, the *size* function is defined as follows. If v is an integer literal $c \in \mathbf{Int}$ then $\mathit{size}(h, c) = c$. If it is a boolean $c \in \mathbf{Bool}$ then $\mathit{size}(h, c) = 0$. If it is a pointer p then:

$$\mathit{size}(h[p \mapsto (j, C\ \overline{v})], p) \overset{\text{def}}{=} 1 + \sum_{i \in \mathit{RecPos}(C)} \mathit{size}(h, v_i)$$

3.1.1. Upper bounds for m and s

Upper bounds to cost and sizes are represented by numbers in $\mathbb{R}^+_\infty \overset{\text{def}}{=} \mathbb{R}^+ \cup \{+\infty\}$. The special value $+\infty$ denotes the absence of upper bounds (either because there are no such bounds, or because the algorithm is not able to infer them).

Our space consumption analysis bounds the m and s components of the resource vector by means of functions that depend on the size of the input. So, memory heap and stack needs are represented as members of the following set, where n denotes the number of parameters of the function being analysed:

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

The notation $D^\perp$ denotes the set $D \cup \{\perp\}$. Throughout this work we use the special value $\perp$ to make undefinedness of functions explicit. If $\xi \in \mathbb{F}$, we say that ξ is undefined for some sizes $\overline{x}$ if and only if $\xi\ \overline{x} = \bot$. Hence, the *domain* of a function $\xi \in \mathbb{F}$ is defined as $\mathrm{dom}\ \xi = \{\overline{x} \in (\mathbb{R}^+_\infty)^n \mid \xi\ \overline{x} \neq \bot\}$. The intuitive meaning of a cost function returning $\bot$ for a given size is that the function whose cost is being inferred does not evaluate to any value for that size. The strictness condition of space cost functions demands that the result is undefined if at least one of the arguments is undefined.

For instance, the memory needs of the function definition *head* $(x : xx) = x$ are zero when the size of the input list is greater than two (the size of the singleton list) and undefined otherwise. For the sake of clarity, we shall use curried notation when dealing with functions in $\mathbb{F}$, and λ-notation when defining them. We assume that the functions defined in this way are *strict*, so $\lambda x.x + 1$ should be read as a function giving $\bot$ when $x = \bot$, and $x + 1$ otherwise. We use $\xi, \xi_1, \ldots$ to denote a generic element of $\mathbb{F}$. If we use an element of $\mathbb{F}$ for bounding *heap* memory needs (that is, the m in (δ, m, s)) we shall use $\mu, \mu_1, \ldots$ as metavariables, whereas in the context of *stack* memory needs (the s component) we use $\sigma, \sigma_1, \ldots$

The following guarded notation will be convenient in what follows. Given a boolean function G and $\xi \in \mathbb{F}$, the notation $[G \to \xi]$ denotes the following:

$$[G \to \xi] = \lambda\overline{x}.\begin{cases} \xi\ \overline{x} & \text{if } G\ \overline{x} \text{ holds} \\ \bot & \text{otherwise} \end{cases}$$

By abuse of notation we factor out the $\lambda\overline{x}$ prefix in both G and ξ. For instance, we write $\xi = \lambda xs.[xs \geq 2 \to 0]$ for defining the memory needs of the *head* function given above.

The usual ordering in $\mathbb{R}^+$ can be extended to $(\mathbb{R}^+_\infty)^\bot$ by taking $\bot$ as the bottommost element and $+\infty$ as the topmost one. Similarly, the arithmetic operations $+$ and $*$ can also be extended to $(\mathbb{R}^+_\infty)^\bot$ by assuming that $x + (+\infty) = x * (+\infty) = +\infty$ for all $x \in \mathbb{R}^+_\infty$, and that $y + \bot = y * \bot = \bot$ for all $y \in (\mathbb{R}^+_\infty)^\bot$.

We shall also need two different operators for denoting least upper bounds in $(\mathbb{R}^+_\infty)^\bot$: $\sqcup$ and $\uplus$. The first one ignores undefined values, whereas the second one returns $\bot$ if at least one of the elements to which it is applied is $\bot$. For example, $\sqcup\{2, 5, \bot, 1\} = 5$, but $\uplus\{2, 5, \bot, 1\} = \bot$.

The $+$, $*$, $\sqcup$ and $\uplus$ operators can be trivially extended to $(\mathbb{R}^+_\infty)^\bot$-valued functions in the usual way, as well as the $\leq$ relation. It is easy to see that these operators are $\leq$-monotonic on their arguments, and hence the $\mathbb{F}$ set is closed under these operators. The partially ordered set of space cost functions

$(\mathbb{F}, \sqsubseteq)$ turns out to be a complete lattice, whose bottommost element is $\lambda\overline{x}.\bot$ and the topmost one is $\lambda\overline{x}. + \infty$.

3.1.2. Upper bounds for δ

The δ component deserves special attention, since it is not a single number, but a mapping from heap region identifiers (natural numbers) to integers. We cannot know, at compile time, which region identifiers are used to build DSs during the execution of an expression, since this information is only known at runtime by means of the value environment E, which associates region variables with actual region identifiers. We have to abstract the region identifiers by their abstract counterparts: region type variables.

Assuming the function definition of (3), let us define $R_f = \{\rho_1, \ldots, \rho_m\}$ and $R^*_f = R_f \cup \{\rho^f_{\mathit{self}}\}$. The function's body can only charge space costs to the regions whose type belongs to R_f and to the working region *self* (of type ρ^f_{self}).

Example 7. Given the following function:

$$\mathit{consPair}\ x\ \mathit{xs}\ \mathit{ys}\ @\ r_1\ r_2\ r_3 = ((x : \mathit{xs})@r_1, (x : (\mathit{ys}@r_2))@r_2)@r_3$$

where $r_i :: \rho_i$, an upper bound to its δ component is

$$\Delta = \lambda x\ \mathit{xs}\ \mathit{ys}. \begin{bmatrix} \rho_1 \mapsto 1 \\ \rho_2 \mapsto \mathit{ys} + 1 \\ \rho_3 \mapsto 1 \end{bmatrix}$$

The function builds a pair in region $r_3 :: \rho_3$ and prepends element x to list xs in region $r_1 :: \rho_1$, so it respectively consumes one constructor cell in each region. It also copies list ys in region $r_2 :: \rho_2$ and then prepends x to it so it consumes $\mathit{ys} + 1$ cells in that region. □

Hence, a function that bounds the memory charges in each region belongs to the following set,

$$\mathbb{D}^* \stackrel{\text{def}}{=} \{\Delta : ((\mathbb{R}^+_\infty)^\bot)^n \to (R^*_f \to \mathbb{R}^+_\infty)^\bot \mid \Delta \text{ is monotonic and strict}\}$$

whose elements are called *abstract heaps*. We use variables $\Delta, \Delta_1, \ldots$ to denote functions in this set.

When calling a given function, the charges to its *self* region are not visible from the caller's point-of-view. In these cases we define $\mathbb{D}$ as in the definition

$\mathbb{D}^*$ above, but substituting R_f for R_f^*. Moreover, given an abstract heap $\Delta \in \mathbb{D}^*$, we denote by $\lfloor\Delta\rfloor$ the function $\lambda\overline{x}.(\Delta\ \overline{x})|_{R_f}$, which discards the information regarding the *self* region, so that the result belongs to $\mathbb{D}$.

The $+$, and $\sqcup$ operators and the $\sqsubseteq$ relation on abstract heaps are defined componentwise, as usual. The same applies to the guarded notation $[G \to \Delta]$. Slightly different is a multiplication operator $*$ we will use later, which is defined in $\mathbb{F} \times \mathbb{D}$.

$$\xi * \Delta = \lambda\overline{x}.[\Delta\ \overline{x} \neq \bot \to \lambda\rho \in R_f.(\xi\ \overline{x}) * (\Delta\ \overline{x}\ \rho)]$$

Finally, we introduce the $\|\cdot\|$ operator, which adds the charges made to all region types into a single cost function:

$$\|\Delta\| = \lambda\overline{x}. \left[\Delta\ \overline{x} \neq \bot \to \sum_{\rho\in R_f} \Delta\ \overline{x}\ \rho\right]$$

Notice the similarity with $\|\delta\|$ notation defined in (2) (in page 19). Let us illustrate these definitions with some examples.

Example 8. Given the set $R_f = \{\rho_1, \rho_2, \rho_3\}$, let us define

$$\Delta_1 = \lambda x. \begin{bmatrix} \rho_1 & \mapsto & x+1 \\ \rho_2 & \mapsto & 2 \\ \rho_3 & \mapsto & x^2+3 \end{bmatrix} \qquad \Delta_2 = [x \geq 2 \to \Delta_1] \qquad \xi = \lambda x.2x$$

Then $\Delta_2 \sqsubseteq \Delta_1$ holds, but $\Delta_1 \sqsubseteq \Delta_2$ does not. Moreover:

$$\Delta_1 + \Delta_2 = \lambda x. \left[x \geq 2 \to \begin{bmatrix} \rho_1 & \mapsto & 2x+2 \\ \rho_2 & \mapsto & 4 \\ \rho_3 & \mapsto & 2x^2+6 \end{bmatrix}\right]$$

$$\xi * \Delta_1 = \lambda x. \begin{bmatrix} \rho_1 & \mapsto & 2x^2+2x \\ \rho_2 & \mapsto & 4x \\ \rho_3 & \mapsto & 2x^3+6x \end{bmatrix} \qquad \|\Delta_2\| = \lambda x.[x \geq 2 \to x^2+x+6]$$

□

In a similar way to cost functions, the ordered set of abstract heaps $(\mathbb{D}, \sqsubseteq)$ is a complete lattice, having $\lambda\overline{x}.\bot$ and $\lambda\overline{x}.\lambda\rho \in R_f. + \infty$ as its bottom- and topmost elements, respectively.

3.1.3. Function signature

Note that all these sets ($\mathbb{F}$, $\mathbb{D}$, and $\mathbb{D}^*$) are parametric with respect to the function definition being analysed. In particular, they depend on its number of data parameters, and the set R_f of region types in its type signature. At some points we shall make the name of the function explicit when referring to these sets, for instance, as in $\mathbb{F}_f$.

These definitions allow us to set up a correspondence between the actual memory consumption of a function, given by a resource vector (δ, m, s), and the results of our analysis:

Definition 2. A *function signature* for f is a triple (Δ, μ, σ), where Δ belongs to $\mathbb{D}_f$, and μ, σ belong to $\mathbb{F}_f$.

The Δ is meant to be an upper bound to δ, where as μ and σ are upper bounds to m and s, respectively. In order to simplify the notation and the examples, from now on we will omit the λ bindings and we will consider them as implicit in the cost expressions, e.g. we will write n instead of $\lambda\overline{x}.n$.

3.2. Abstract interpretation

So far we have defined *what* a signature is. In this section we show *how* to infer a correct signature for a given function definition. Given an expression e, our aim is to find a tuple (Δ, μ, σ) which is an upper bound to its memory consumption, provided we know the signatures of the functions that are invoked from this expression.

3.2.1. Basic expressions

We define an abstract interpretation as a set of rules in Figure 9. The interpretation of a basic expression $[\![be]\!]\ \Sigma\ \Gamma\ td$ receives a signature environment Σ, which maps each function name g appearing in e to its signature $(\Delta_g, \mu_g, \sigma_g)$. It also receives a typing environment Γ, giving the type of each region variable in scope, and the statically determined length td of the runtime environment when calling a function application. The latter has the same meaning as in the resource-aware semantics of Figure 5. The interpretation is also parametric on the function definition f being inferred. However, and to avoid excessive subscripting, we consider this definition fixed, so it is left out in the rules.

If x is a variable, the notation $|x|$ denotes the size function associated with x, which belongs to $\mathbb{F}$ and is assumed to have been inferred by an external size analysis.

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

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

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

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

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

Figure 9: Abstract interpretation for basic expressions.

If R_f is the set of the context function's region parameters, the $[\,]$ notation stands for the abstract heap $\lambda\overline{x}.\lambda\rho.0$ where $\rho \in R_f$, and whenever none of the x_i is $\bot$. Similarly, the binding $[\rho' \mapsto \xi]$ denotes the abstract heap that yields $\xi\ \overline{x}$ for ρ' and 0 for every $\rho \neq \rho'$.

Notice the similarity between the results of the abstract interpretation in Figure 9 and the resource vectors of the semantic rules in Figure 5. The only rule worth explaining is that of function application. Given a function definition $g\ \overline{y}\ @\ \overline{r}' = e_g$, 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 Σ, so let $(\Delta_g, \mu_g, \sigma_g) = \Sigma(g)$. Each component is a function which depends on the sizes of its parameters $\overline{y}$, so we have to pass the sizes of the actual arguments $|a_i|$, which, in turn, are functions of the sizes $\overline{x}$ 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 Δ_g, μ_g or σ_g.

Notice also that, with regard to the computation of Δ, the type of the arguments in the function application are instances of the most general function's type. This means that if several RTVs of g's type are mapped to the same RTV ρ, the charges done to ρ are the sum of the charges made by g to the former RTVs.

Example 9. Recall function *consPair* defined in Example 7, where $\Delta = [\rho_1 \mapsto 1, \rho_2 \mapsto ys + 1, \rho_3 \mapsto 1]$. The application *consPair y us zs*@$r\ r\ r'$, where $r :: \rho$ and $r' :: \rho'$, would consume $[\rho \mapsto |zs| + 2, \rho' \mapsto 1]$. □

Hence we have to find a correspondence between the region types of g

and the region types of the particular instance used in the call. That is what the function *unify* does.

Definition 3. Given a type environment Γ, a function name $g \in \mathbf{Fun}$ and a sequence of region variables $\overline{r}$, we say that $\theta = \mathit{unify}(\Gamma, g, \overline{r})$, iff $\Gamma(g) = \forall \overline{\alpha \rho}. \overline{t} \rightarrow \overline{\rho} \rightarrow t$, and $\forall j \in \{1..|\overline{\rho}|\}.\theta(\rho_j) = \Gamma(r_j)$.

If there are several RTVs of g's type being mapped to the same RTV ρ in the function application, then the charges done to the region of type ρ are the sum of the charges made by g to the RTVs ρ' 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 $\rho \in R_f \cup \{\rho_{\mathit{self}}\}$.

Example 10. In Example 9, we obtain $\theta = [\rho_1 \mapsto \rho, \rho_2 \mapsto \rho, \rho_3 \mapsto \rho']$. If, for example, the size of the third argument were $|\mathit{zs}| = \mathit{vs} - 1$, being *vs* an argument of the caller, the application would consume: $[\rho \mapsto (\mathit{vs}-1)+2, \rho' \mapsto 1]$, i.e. $[\rho \mapsto \mathit{vs} + 1, \rho' \mapsto 1]$. □

Back to our abstract interpretation rule for function applications, so far we have obtained a tuple $(\theta \downarrow_{\overline{y}} \Delta_g, \mu \; \overline{y}, \sigma \; \overline{y})$, which is an upper bound to the memory costs of executing the function's *body*. From these we can easily compute the costs of the function application itself: we just have to proceed as in rule [*App*] of Figure 5.

In the following we shall use the notation $[\![\mathit{be}]\!]_\Delta \; \Sigma \; \Gamma \; \mathit{td}$, $[\![\mathit{be}]\!]_\mu \; \Sigma \; \Gamma \; \mathit{td}$ and $[\![\mathit{be}]\!]_\sigma \; \Sigma \; \Gamma \; \mathit{td}$ to refer to the first, second and third components of $[\![\mathit{be}]\!] \; \Sigma \; \Gamma \; \mathit{td}$ respectively. We shall omit the *td*, Γ and Σ when they are not relevant, or can be deduced from the context.

3.2.2. Compound expressions

Our next goal is the inference of heap space consumption of compound expressions (**let** and **case**). This works as follows: an expression e is transformed into a set of *sequences* of basic expressions. Each sequence represents a possible execution flow of the expression, and the whole set of sequences captures all the possible execution flows that may arise when executing this expression. This transformation destroys the structure of the program: instead of having nested expressions, the result comprises all the basic expressions being executed into a single "level" of nesting (that is why this transformation will be called *flattening*). However, this does not affect heap space consumption, since the latter does not depend on the structure of the expression. The following example provides an intuition on this fact.

Example 11. Given the following expressions,

$$\begin{array}{rcl} e & \equiv & \mathbf{let}\ x_1 = be_1\ \mathbf{in}\ (\mathbf{let}\ x_2 = be_2\ \mathbf{in}\ be_3) \\ e' & \equiv & \mathbf{let}\ x_2 = (\mathbf{let}\ x_1 = be_1\ \mathbf{in}\ be_2)\ \mathbf{in}\ be_3 \end{array}$$

let (δ_i, m_i, s_i) the resource vector associated to the execution of be_i for $i \in \{1..3\}$. If $x_1 \notin fv(be_3)$ and we can execute e, we can also execute e'. If we compare the δ components of each, we obtain $\delta_1 + (\delta_2 + \delta_3)$ and $(\delta_1 + \delta_2) + \delta_3$ respectively. These are trivially equal. If we compare the m component of their resource vectors, it can be shown (by using that $\|\delta_1 + \delta_2\| = \|\delta_1\| + \|\delta_2\|$) that both are equal to the following expression:

$$\sqcup\{m_1, \|\delta_1\| + m_2, \|\delta_1\| + \|\delta_2\| + m_3\} \tag{4}$$

In both e and e', the expression be_1 is the first to be executed, then be_2 and be_3 come in the last place. That is why we can represent both expressions as a single sequence $[be_1, be_2, be_3]$. □

When considering sequences of expressions we lose the pattern guards of the **case** expressions. Nevertheless, these guards provide useful information on the size of the DS being matched against. For instance, assume the expression **case** x **of** $(x : xx) \to e_2$. If the branch e_2 is executed, we know for sure that the size of x must be greater or equal than 2. In general, this size information will be included in the execution sequences as *guards*, specifying under which sizes of the **case** discriminant the execution of each sequence may take place. This motivates the following definition.

Definition 4. A *sequence* is a list of basic expressions $be_1, \ldots, be_n$ preceded by a boolean function G. We use $[G \to be_1, ...be_n]$ to denote sequences. The notation $[be_1, \ldots, be_n]$ stands for the sequence $[\mathit{true} \to be_1, \ldots, be_n]$.

We define the concatenation ($+\!\!+$) of sequences as follows:

$$[G_1 \to be_1, \ldots, be_n] +\!\!+ [G_2 \to be'_1, \ldots, be'_m] = [G_1 \wedge G_2 \to be_1, \ldots, be_n, be'_1, \ldots, be'_m]$$

$$
\begin{array}{lcl}
\mathit{seqs}\ \mathit{be} & = & \{[\mathit{be}]\} \\
\mathit{seqs}\ (\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2) & = & \{\mathit{seq}_1 +\!\!+\ \mathit{seq}_2 \mid \mathit{seq}_1 \in \mathit{seqs}\ e_1, \mathit{seq}_2 \in \mathit{seqs}\ e_2\} \\
\mathit{seqs}\ (\mathbf{case}\ x\ \mathbf{of}\ \overline{\mathit{alt}}) & = & \bigcup_{r=1}^{n}(|x| \geq 1 + |\mathit{RecPos}(C_r)| \to \mathit{seqs}\ e_r) \\
 & & \mathbf{where}\ n = |\overline{\mathit{alt}}| \\
 & & \qquad\quad \mathit{alt}_r = C_r\ \overline{x_r} \to e_r
\end{array}
$$

Figure 10: Definition of *seqs*.

```
$insert :: Int \to [Int]@\rho_1 \to \rho_2 \to [Int]@\rho_2$
$insert\ y\ zs\ @\ r =$
    $\mathbf{case}\ zs\ \mathbf{of}$
        $[\,] \to \mathbf{let}\ x_1 = [\,]\ @\ r\ \mathbf{in}\ (y : x_1)@r$
        $(x : xx) \to \mathbf{let}\ b = y \leq x$
                  $\mathbf{in\ case}\ b\ \mathbf{of}$
                      $True \to \mathbf{let}\ x_2 = (x : xx)@r\ \mathbf{in}\ (y : x_2)@r$
                      $False \to \mathbf{let}\ x_3 = insert\ y\ xx\ @\ r\ \mathbf{in}\ (x : x_3)@r$

$insSort :: [Int]@\rho_1 \to \rho_2 \to [Int]@\rho_2$
$insSort\ zs\ @\ r =$
    $\mathbf{case}\ zs\ \mathbf{of}$
        $[\,] \to [\,]\ @\ r$
        $(x : xx) \to \mathbf{let}\ xx' = insSort\ xx\ @\ r\ \mathbf{in}\ insert\ x\ xx'\ @\ r$
```

Figure 11: *Core-Safe* code of insertion sort algorithm.

If *seq* is a sequence, we can strengthen its guard with the notation $G \to \mathit{seq}$, which stands for $G \to [G_1 \to \mathit{be}_1, \ldots, \mathit{be}_n] = [G \wedge G_1 \to \mathit{be}_1, \ldots, \mathit{be}_n]$, and is extended to *sets* of sequences in the trivial way.

The function *seqs*, shown in Figure 10, transforms an expression into a set of sequences representing all the possible execution paths. In order to transform a **let** expression, *seqs* gathers all the sequences of the auxiliary expression e_1 and the main expression e_2 and considers all the combinations. With respect to **case** expressions, it collects the sequences of each branch and adds the corresponding size guard imposed by the recursive positions of the pattern: the size of the discriminant must be one plus the number of recursive positions of its constructor.

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

$$
\begin{array}{lcl}
[\![[G \to \mathit{be}_1, ..., \mathit{be}_n]]\!]_\Delta & \stackrel{\text{def}}{=} & [G \to [\![\mathit{be}_1]\!]_\Delta + \ldots + [\![\mathit{be}_n]\!]_\Delta] \\
[\![[G \to \mathit{be}_1, ..., \mathit{be}_n]]\!]_\mu & \stackrel{\text{def}}{=} & [G \to \biguplus\{\ [\![\mathit{be}_1]\!]_\mu, \\
 & & \qquad \|[\![\mathit{be}_1]\!]_\Delta\| + [\![\mathit{be}_2]\!]_\mu, \\
 & & \qquad \ldots, \\
 & & \qquad \sum_{j=1}^{n-1} \|[\![\mathit{be}_j]\!]_\Delta\| + [\![\mathit{be}_n]\!]_\mu\}]
\end{array}
$$

Figure 12: Abstract interpretation compound expressions and sequences.

Example 12. Let us consider the *insertion sort* algorithm of Figure 1. The *Core-Safe* code of the two functions involved is shown in Figure 11. We get:

$$
\mathit{seqs}\ e_{\mathit{insert}} = \left\{ \begin{array}{l} \mathit{seq}_1 \equiv [\mathit{zs} \geq 1 \to [\,]\ @\ r, (y : x_1)@r] \\ \mathit{seq}_2 \equiv [\mathit{zs} \geq 2 \to y \leq x, (x : \mathit{xx})@r, (y : x_2)@r] \\ \mathit{seq}_3 \equiv [\mathit{zs} \geq 2 \to y \leq x, \mathit{insert}\ y\ \mathit{xx}\ @\ r, (x : x_3)@r] \end{array} \right\}
$$

$$
\mathit{seqs}\ e_{\mathit{insSort}} = \left\{ \begin{array}{l} \mathit{seq}_4 \equiv [\mathit{zs} \geq 1 \to [\,]\ @\ r] \\ \mathit{seq}_5 \equiv [\mathit{zs} \geq 2 \to \mathit{insSort}\ \mathit{xx}\ @\ r, \mathit{insert}\ x\ \mathit{xx}'\ @\ r] \end{array} \right\}
$$

□

The extension of the abstract interpretation rules to compound expressions will be done in terms of their decomposition into sequences of basic expressions. Example 11 gave us an intuition on how to do this. If we replace the δ_i and m_i components by their abstract counterparts Δ_i and μ_i, and generalize the formula given in (4) we get the definitions of Figure 12. The reason of choosing $\uplus$ instead of $\sqcup$ is that, if the cost of one of the basic expressions is undefined for some input sizes, we want to cancel out the whole sequence. In case a compound expression leads to several sequences, we have to take the least upper bound of all of them, as shown in Figure 12.

Example 13. Back to our insertion sort algorithm, assume a signature Σ in which $\Sigma(\mathit{insert}) = (f_\Delta, f_\mu, \_)$ and $\Sigma(\mathit{insSort}) = (g_\Delta, g_\mu, \_)$ for some $f_\Delta \in \mathbb{D}_{\mathit{insert}}$, $f_\mu \in \mathbb{F}_{\mathit{insert}}$, $g_\Delta \in \mathbb{D}_{\mathit{insSort}}$, $g_\mu \in \mathbb{F}_{\mathit{insSort}}$ that are defined when $\mathit{zs} \geq 1$.

Let us apply the $[\![\cdot]\!]_\Delta$ and $[\![\cdot]\!]_\mu$ interpretations to the first two sequences under an environment $\Gamma = [r \mapsto \rho_2]$:

$$\begin{array}{l}
[\![seq_1]\!]_\Delta = [zs \geq 1 \rightarrow [\rho_2 \mapsto 2, \rho_{self} \mapsto 0]] \\
[\![seq_2]\!]_\Delta = [zs \geq 2 \rightarrow [\rho_2 \mapsto 2, \rho_{self} \mapsto 0]]] \\
[\![seq_1]\!]_\mu = [zs \geq 1 \rightarrow 2] \\
[\![seq_2]\!]_\mu = [zs \geq 2 \rightarrow 2]
\end{array}$$

Regarding the third sequence we assume that $|xx| = [zs \geq 2 \rightarrow zs-1]$ and that $|y| = y$. We get the guard $G = zs \geq 1$, which is used in the respective interpretations of the *insert* function application:

$$\begin{array}{lcl}
[\![insert\ y\ xx\ @\ r]\!]_\Delta & = & \left[zs \geq 2 \rightarrow \left[\begin{array}{lll} \rho_2 & \mapsto & f_\Delta\, y\, (zs-1)\, \rho_2 \\ \rho_{self} & \mapsto & 0 \end{array}\right]\right] \\
[\![insert\ y\ xx\ @\ r]\!]_\mu & = & [zs \geq 2 \rightarrow f_\mu\, y\, (zs-1)]
\end{array}$$

Therefore:

$$\begin{array}{lcl}
[\![seq_3]\!]_\Delta & = & \left[zs \geq 2 \rightarrow \left[\begin{array}{lll} \rho_2 & \mapsto & 1 + f_\Delta\, y\, (zs-1)\, \rho_2 \\ \rho_{self} & \mapsto & 0 \end{array}\right]\right] \\
[\![seq_3]\!]_\mu & = & [zs \geq 2 \rightarrow \sqcup\{f_\mu\, y\, (zs-1), f_\Delta\, y\, (zs-1)\, \rho_2 + 1\}]
\end{array}$$

Finally, let us assume that $|xx| = |xx'| = [zs \geq 2 \rightarrow zs-1]$. We get:

$$\begin{array}{l}
[\![seq_4]\!]_\Delta = [zs \geq 1 \rightarrow [\rho_2 \mapsto 1, \rho_{self} \mapsto 0]] \\
[\![seq_5]\!]_\Delta = \left[zs \geq 2 \rightarrow \left[\begin{array}{lll} \rho_2 & \mapsto & g_\Delta\, (zs-1)\, \rho_2 + f_\Delta\, (|x|\ zs)\, (zs-1)\, \rho_2 \\ \rho_{self} & \mapsto & 0 \end{array}\right]\right] \\
\\
[\![seq_4]\!]_\mu = [zs \geq 1 \rightarrow 1] \\
[\![seq_5]\!]_\mu = [zs \geq 2 \rightarrow \sqcup\{g_\mu\, (zs-1), g_\Delta\, (zs-1) + f_\mu\, (|x|\ zs)\, (zs-1)\}]
\end{array}$$

Notice that the $[\![\cdot]\!]_\mu$ interpretation of the last sequence depends on the size of the variable x. A sensible size analysis would be expected to give reasonable bounds to the sizes of xx and xx', but we cannot expect the same for x, since it stands for an element of the input list. That is why we left the symbolic application $(|x|\ zs)$ in the result. As we will see later, if the f_Δ function does not depend on its first argument this application is ignored. $\square$

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

Figure 13: Stack consumption of compound expressions.

Example 14. Following Example 13 we get:

$$\begin{array}{lcl}
[\![e_{insert}]\!]_\Delta & = & [zs \geq 1 \to [\rho_2 \mapsto 2, \rho_{self} \mapsto 0]] \sqcup \\
 & & [zs \geq 2 \to [\rho_2 \mapsto 1 + f_\Delta\ y\ (zs-1)\ \rho_2, \rho_{self} \mapsto 0]] \\
 & & \\
[\![e_{insert}]\!]_\mu & = & [zs \geq 1 \to 2] \sqcup \\
 & & [zs \geq 2 \to \sqcup\{f_\mu\ y\ (zs-1), f_\Delta\ y\ (zs-1) + 1\}] \\
 & & \\
[\![e_{insSort}]\!]_\Delta & = & [zs \geq 1 \to [\rho_2 \mapsto 1, \rho_{self} \mapsto 0]] \sqcup \\
 & & \left[zs \geq 2 \to \left[\begin{array}{ll} \rho_2 & \mapsto\ g_\Delta\ (zs-1)\ \rho_2 + f_\Delta(|x|\ zs)\ (zs-1)\ \rho_2 \\ \rho_{self} & \mapsto\ 0 \end{array}\right]\right] \\
 & & \\
[\![e_{insSort}]\!]_\mu & = & [zs \geq 1 \to 1] \sqcup \\
 & & [zs \geq 2 \to \sqcup\{g_\mu\ (zs-1), g_\Delta\ (zs-1) + f_\mu(|x|\ zs)\ (zs-1)\}]
\end{array}$$

□

In order to bound stack costs, we cannot apply the flattening-based approach, since this transformation breaks the structure of an expression and, unlike heap costs, the stack costs *do* depend on this structure. Instead of defining the $[\![\cdot]\!]_\sigma$ interpretation as a function of the sequences originated from a compound expressions, we define it compositionally as in Figure 13, which roughly ressembles its counterpart in the semantics of Figure 5.

Example 15. The $[\![\cdot]\!]_\sigma$ interpretation of our running example e_{insert} leads to the following results with $td = 3$, if we assume that $\Sigma(insert) = (\_, \_, f_\sigma)$ such that dom $f_\sigma = \{(y, zs) \mid zs \geq 1\}$, and $\Sigma(insSort) = (\_, \_, g_\sigma)$ such that dom $g_\sigma = \{zs | zs \geq 1\}$:

$$\begin{array}{lcl}
[\![e_{insert}]\!]_\sigma\ \Sigma\ 3 & = & [zs \geq 1 \to 3] \sqcup \\
 & & [zs \geq 2 \to 8 + f_\sigma\ y\ (zs-1)] \\
[\![e_{insSort}]\!]_\sigma\ \Sigma\ 2 & = & [zs \geq 1 \to 1] \sqcup \\
 & & [zs \geq 2 \to \sqcup\{1 + f_\sigma\ (|x|\ zs)\ (zs-1), 6 + g_\sigma\ (zs-1)\}]
\end{array}$$

□

It is easy to show (see [35]) that the results of the interpretation of an expression fall into $\mathbb{F}$ and $\mathbb{D}^*$, respectively.

3.3. Correctness of the abstract interpretation

In this section we aim to prove that the tuple (Δ, μ, σ) resulting from the abstract interpretation of e is an upper bound to the actual resource vector (δ, m, s) resulting from the execution of e. Firstly we have to make precise what do we mean with (Δ, μ, σ) being an upper bound to (δ, m, s). As a first step, we define this notion in the context of function definitions, rather than isolated expressions. Given the function definition $f\ \overline{x}\ @\ \overline{r} = e_f$, assume the following $\Downarrow$-judgement of its body e_f, whose derivation is referred to as the *context derivation*:

$$E_0 \vdash h_0, k_0, td_0, e_f \Downarrow h_f, k_0, v_f, (\delta_f, m_f, s_f) \tag{5}$$

In the correctness theorem shown below, we will assume that we have correct signatures for all the functions called from e_f.

We are particularly interested in the part of the execution occurring under the same call context as (5). That is, those $\Downarrow$-judgements that belong to the derivation of (5), but not to the derivation of any further evaluation of e_f contained within (5). If the function is nonrecursive any subderivation meets this requirement. Otherwise only the parts corresponding to the main call are considered and the correctness result we present is this section requires having a correct signature for the function itself.

We shall characterize such judgements by means of their region counter, which will be always equal to k_0, and the expression being evaluated, which is a subexpression of e_f.

Intuitively, we expect a correct signature (Δ, μ, σ) to be an upper bound to the actual vector (δ_f, m_f, s_f) for every input size. In order to check whether this fact holds, we need to know the sizes $\overline{s}$ of the input parameters, since the components of a signature are functions on these sizes.

Definition 5. Given a sequence of sizes $\overline{s}$ of the input arguments of the context function f we say that μ is an upper bound to m in the context of $\overline{s}$ (denoted $\mu \succeq_{\overline{s}} m$) iff $\mu\ \overline{s} \geq m$. Analogously, σ is an upper bound to s in the context of $\overline{s}$ (denoted $\sigma \succeq_{\overline{s}} s$) iff $\sigma\ \overline{s} \geq s$.

However, the case of Δ is more involved, since we have to take into account the correspondence between the static RTVs $\rho_1 \dots \rho_m$ and the dynamic region identifiers given by $E_0(r_1) \dots E_0(r_m)$. This correspondence is made explicit via *region instantiations*. A region instantiation η is a function from RTVs to natural numbers (interpreted as region identifiers), and it is demanded to be *consistent* with the information provided by the runtime environment E_0 and the types of the region parameters, given by a typing environment Γ_0 which contains the bindings $[r_j : \rho_j]$ for each $j \in \{1..m\}$. For the purposes of this paper, we assume that Γ_0 is injective, so η can be considered as an abbreviation for $(E_0 \circ \Gamma_0^{-1})|_{R_f^*}$.

The key role of η bindings is the following: if there are several ρ's being mapped by η to the same region number i, the sum of the estimations of each individual ρ is an upper estimation of the global amount of charges done to the i-th region.

Definition 6. Given a sequence of sizes $\overline{s}$ of the input arguments of the context function f, a number k of regions and a region instantiation η, we say that Δ is an upper bound to δ in the context of $\overline{s}$, k and η (denoted $\Delta \succeq_{\overline{s},k,\eta} \delta$) iff

$$\forall j \in \{0..k\}. \sum_{\substack{\rho \in R_f^* \\ \eta(\rho)=j}} \Delta\ \overline{s}\ \rho \geq \delta(j)$$

Example 16. Assume a function definition $f\ x\ y\ @\ r_1\ r_2 = e_f$ and the execution of its body e_f under a environment E and a heap h with three regions (that is, $k = 2$). We also assume that $s_x = \mathit{size}(h, E(x)) = 4$, $s_y = \mathit{size}(h, E(y)) = 2$ and that the function is typable under an environment Γ such that $\Gamma(r_i) = \rho_i$ for $i \in \{1, 2\}$. If $E(r_1) = 0$, $E(r_2) = 0$ and $E(\mathit{self}) = 1$, then we get $\eta = [\rho_1 \mapsto 0, \rho_2 \mapsto 0, \rho_{\mathit{self}} \mapsto 1]$. Let us define Δ as follows:

$$\Delta = \begin{bmatrix} \rho_1 & \mapsto & 2x + y \\ \rho_2 & \mapsto & xy \\ \rho_{\mathit{self}} & \mapsto & 7x + 5 \end{bmatrix}$$

Then Δ is a correct upper bound to $\delta = [0 \mapsto 10, 1 \mapsto 33]$ in the context of s_x, s_y, k and η, since $\Delta\ 4\ 2 = [\rho_1 \mapsto 10, \rho_2 \mapsto 8, \rho_3 \mapsto 33]$, and it holds that $10 + 8 \geq \delta(0)$ and $33 \geq \delta(1)$. $\square$

Given these definitions, we are ready to give a formal notion of a function signature (Δ, μ, σ) being correct:

Definition 7 (Correct signature). Let (Δ, μ, σ) be the signature of a function definition $f\ \overline{x}\ @\ \overline{r} = e_f$ and Γ the type environment inferred for e_f. This signature is said to be *correct* if for all h, h', k, E_f, $\overline{v}$, $\overline{\imath}$, v, δ, m, s, $\overline{s}$, η such that

1. $E_f \vdash h, k+1, e_f \Downarrow h', k+1, v, (\delta, m, s)$
   where $E_f = [\overline{x} \mapsto \overline{v}, \overline{r} \mapsto \overline{\imath}, \mathit{self} \mapsto k+1]$
2. For each $j \in \{1..|\overline{s}|\}$, $s_j = \mathit{size}(h, v_j)$
3. η is the consistent region instantiation determined by E_f and Γ

then $\Delta \succeq_{\overline{s},k,\eta} \delta|_k$, $\mu \succeq_{\overline{s}} m$, and $\sigma \succeq_{\overline{s}} s$.

Notice that all these $\succeq$ relations are parametric with respect to the sizes of the input arguments. That is why we cannot tell, for an arbitrary e, whether the result of $[\![e]\!]$ is an upper bound to its associated resource vector (δ, m, s), unless we consider the execution of this expression in the context derivation specified in (5). Besides this, the result of the abstract interpretation depends on the function signatures contained within Σ, with are assumed to be correct, and the size functions $|\cdot|$, which are assumed to be exact or upper approximations of the actual runtime sizes of its corresponding DSs. A precise notion of a correct size analysis can be found in [35]. Roughly speaking, a size function $|x|$ is correct if it is always greater that the size of the data structure pointed to by x at runtime.

Theorem 1 (Correctness of the abstract interpretation). *Let us assume a context function definition $f\ \overline{x}\ @\ \overline{r} = e_f$, the inferred global type environment Γ for e_f, a Σ containing correct signatures for all the functions called from e_f, an initial environment E_0 and a heap h_0 such that the judgement* (5) *is derivable. For each subexpression e of e_f and E, td, h, h', v, δ, m, s such that*

1. *For every $seq \in seqs\ e$, every occurrence of $|\cdot|$ in the evaluation of $[\![seq]\!]_\Delta\ \Sigma\ \Gamma$ and $[\![seq]\!]_\mu\ \Sigma\ \Gamma$ has been inferred with a correct size analysis*
2. *$E \vdash h, k_0, td, e \Downarrow h', k_0, v, (\delta, m, s)$ belongs to the derivation of* (5)

then

1. $[\![e]\!]_\sigma\ \Sigma\ td \succeq_{\overline{s}} s$
2. *There exists some $seq \in seqs\ e$ such that $[\![seq]\!]_\Delta\ \Sigma\ \Gamma \succeq_{\overline{s},k_0,\eta} \delta$ and $[\![seq]\!]_\mu\ \Sigma\ \Gamma \succeq_{\overline{s}} m$*

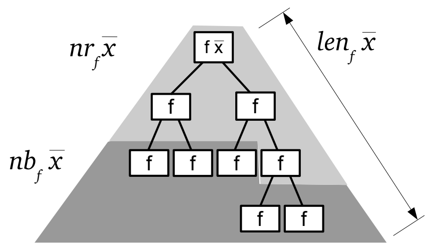


Figure 14: Representation of the activation tree of a function call to f.

where $\forall j \in \{1..|\overline{s}|\}.s_j = size(h, E_0(x_j))$*, and* η *is the consistent region instantiation determined by* E *and* Γ*.*

Proof. By induction on the structure of e.

From this correctness result we can devise a way for inferring upper bounds to the heap and stack consumption of a non-recursive function, assuming that the functions called from it have already been inferred. We just have to apply the abstract interpretation to the body of the function definition. The inference of recursive function definitions is far more involved, and its study is deferred to the following section.

4. Memory consumption of recursive function definitions

The obvious question that arises when applying the abstract interpretation to a recursive function is which signature associated to f must be stored into the signature environment Σ. The result of the abstract interpretation will be correct if this initial signature is correct, but this signature is precisely what we aim to infer. We have to find a correct upper bound to the memory consumption of a function definition by other means different from the $[\![\cdot]\!]$-interpretations. Once these upper bounds are computed, it still makes sense to apply the abstract interpretation under a signature environment Σ containing those bounds. The new results are also correct, by Theorem 1, but they might be more precise than the initial ones, as shown in [34, 35].

This section is devoted to the computation of these *initial approximations*, which will be called Δ_0, μ_0 and σ_0. In order to compute them, we need some information regarding the number of recursive calls generated during the evaluation of a call to f. This information will be given as a function of the input sizes $\overline{x}$. We represent these recursive calls by means of activation trees (or *call trees*), as shown in Figure 14. Before computing an upper bound to the memory costs, we assume that the following information is available as elements of $\mathbb{F}$:

nr_f Upper bound to the number of calls to f which invoke f again. This number corresponds to the internal nodes of f's call tree.

nb_f Upper bound to the number of basic calls to f that do not invoke f again. It corresponds to the leaf nodes of f's call tree.

len_f Upper bound to the maximum length of f's call chains. It corresponds to the height of f's call tree.

In general these functions are not independent of each other. For instance, with linear recursion we 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*, and we can use the techniques described in [28] for computing them. Another possibility is to give definitions of these components as recurrence relations and obtain closed forms by using recurrence solving tools, such as PUBS [5, 6], possibly in combination with polynomial interpolation-based techniques [36].

Example 17. Assume a call to *insert y zs* where $zs = [z_1, \ldots, z_n]$. In the worst case we get the following sequence of calls:

$$\begin{array}{lll}
 & insert\ y\ [z_1, \ldots, z_n] & \text{1st call} \\
\to & insert\ y\ [z_2, \ldots, z_n] & \text{2nd call} \\
\vdots & & \\
\to & insert\ y\ [z_n] & n\text{-th call} \\
\to & insert\ y\ [\,] & (n+1)\text{-th call}
\end{array}$$

So we obtain $n+1$ calls to *insert*, n of which are recursive. Since the size of a list is its number of elements plus one, we get the following functions:

$$\begin{array}{rcl}
nb_{insert} & = & 1 \\
nr_{insert} & = & [zs \geq 1 \to zs - 1] \\
len_{insert} & = & zs
\end{array}$$

The sequence of calls for *insSort zs* is the following

$$\begin{array}{lll} & \mathit{insSort}\ [z_1, \ldots, z_n] & \text{1st call} \\ \to & \mathit{insSort}\ [z_2, \ldots, z_n] & \text{2nd call} \\ \vdots & & \\ \to & \mathit{insSort}\ [z_n] & n\text{-th call} \\ \to & \mathit{insSort}\ [\,] & (n+1)\text{-th call} \end{array}$$

and consequently we get the following functions:

$$\begin{array}{rcl} nb_{\mathit{insSort}} & = & 1 \\ nr_{\mathit{insSort}} & = & [zs \geq 1 \to zs - 1] \\ len_{\mathit{insSort}} & = & zs \end{array}$$

□

In the following subsections we shall present three algorithms for computing our initial Δ_0, μ_0 and σ_0 (4.1.1, 4.1.2 and 4.1.3, respectively), and prove that they are correct bounds to the actual memory needs of the program.

4.1. Initial approximations: Δ_0, μ_0, σ_0

4.1.1. Algorithm for computing Δ_0

In order to grasp the intuitive meaning of the algorithm, let us consider the activation tree corresponding to a function call to $f\ \overline{x}\ @\ \overline{r}$ shown in Figure 15(a). The grey nodes correspond to base calls, whereas the white ones correspond to recursive calls. We can abstract each node by its charges on memory as follows:

- Base nodes are abstracted by applying the $[\![\cdot]\!]_\Delta$-interpretation to the set of *base sequences* of the function's body, that is, those sequences not containing recursive calls to f. We discard sequences with recursive calls, since their execution cannot take place in a base case. By considering only base sequences we avoid inserting a signature for f in the environment Σ, as it is not relevant. Let us denote by Δ_b the result of applying the interpretation to the set of base sequences.

- Recursive nodes are abstracted by applying the $[\![\cdot]\!]_\Delta$-interpretation to the set of of *recursive sequences* of the function's body, that is, those sequences containing at least one recursive call to f. We are interested in the charges done by each individual white node without taking into

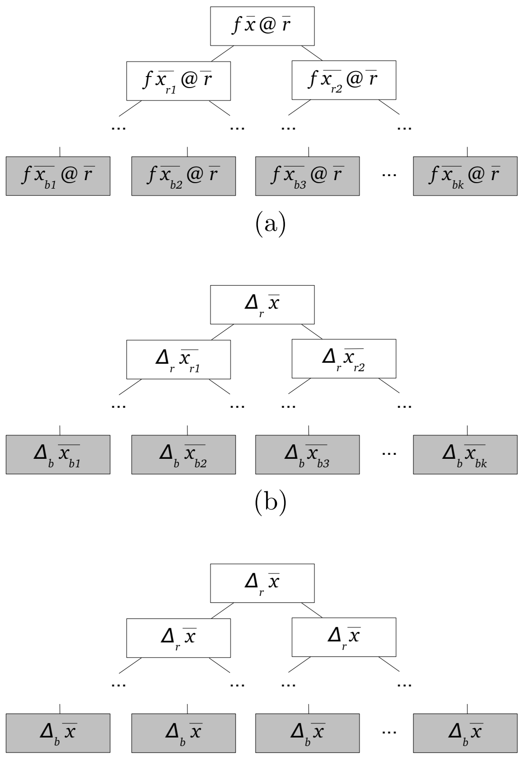


Figure 15: Activation tree corresponding to $f\ \overline{x}\ @\ \overline{r}$ and its abstraction.

$$\begin{array}{llcl}
\multicolumn{4}{l}{\mathit{computeDelta}\ (f\ \overline{x}\ @\ \overline{r} = e_f)\ \Sigma\ \Gamma\ nb\ nr = (\lfloor\Delta_b\rfloor * nb + \lfloor\Delta_r\rfloor * nr) \sqcup \lfloor\Delta_b\rfloor} \\
\quad\textbf{where} & S & = & \mathit{seqs}\ e_f \\
 & (S_b, S_r) & = & \mathit{splitExp}_f\ S \\
 & \Delta_b & = & [\![S_b]\!]_\Delta\ \Sigma\ \Gamma \\
 & \Delta_r & = & [\![S_r]\!]_\Delta\ (\Sigma \uplus [f \mapsto ([\,], 0, 0)])\ \Gamma
\end{array}$$

Figure 16: Algorithm for computing Δ_0.

account its descendants. That is why we attach the empty signature $([], 0, 0)$ to f in the environment Σ. In this way the costs of the recursive calls done by f are ignored. We use Δ_r to denote this abstract heap.

The result of this abstraction is depicted in Figure 15(b). In the internal nodes of the tree we get $\Delta_r\ \overline{x_{ri}}$, where i depends on the particular recursive call. In the leafs we obtain $\Delta_b\ \overline{x_{bi}}$, where i depends on the particular base call. The charges done by the whole tree will be equal to the sum of the charges of the recursive nodes plus the sum of the charges of the base nodes. We can further simplify this abstraction if we assume that the value of Δ_r applied to the size of the arguments in the root call $\overline{x}$ is greater than or equal to the value of Δ_r applied to the arguments of the recursive calls $\overline{x_{r1}}, \overline{x_{r2}}$, etc. When this property applies to Δ_r we can replace each $\Delta_r\ \overline{x_{ri}}$ in each recursive node by $\Delta_r\ \overline{x}$. If it also applies to Δ_b we can proceed with the base nodes in a similar way, so we obtain a tree (see Figure 15(c)) that is an upper approximation to the previous one, but now all the recursive nodes charge the same cost, and so do the base nodes. If nb and nr respectively approximate the number of base and recursive nodes, our initial approximation is given by the function $\lfloor \Delta_b \rfloor * nb + \lfloor \Delta_r \rfloor * nr$.

The property we are assuming on Δ_b and Δ_r usually holds in practice, and it is formally defined as follows:

Definition 8. An abstract heap $\Delta \in \mathbb{D}$ is said to be *parameter-decreasing* with respect to a function definition $f\ \overline{x}\ @\ \overline{r} = e_f$ iff, for every recursive call $f\ \overline{a}\ @\ \overline{r}'$ occurring in e_f, and for all $\overline{x} \in ((\mathbb{R}^+_\infty)^\perp)^n$, it holds that $\Delta\ \overline{x} \geq \Delta\ \overline{s}$, where $\overline{s}$ denotes the size of the parameters in the recursive call, that is, $s_i \overset{\text{def}}{=} |a_i|\ \overline{x}$.

The algorithm for computing the initial approximation Δ_0 is shown in Figure 16. We assume the existence of a function $splitExp_f$ that divides the sequences in S into recursive sequences (S_r) and base sequences (S_b). The least upper bound with $\lfloor \Delta_b \rfloor$ handles those input sizes in which $\lfloor \Delta_r \rfloor$ becomes undefined.

Example 18. In our *insSort* running example, if $S_1 = seqs\ e_{insert}$ and $S_2 = seqs\ e_{insSort}$, defined in Examples 12 and 13, we obtain $splitExp_{insert}\ S_1 = (S_{1b}, S_{1r})$ and $splitExp_{insSort}\ S_2 = (S_{2b}, S_{2r})$ where

$$\begin{array}{lllllll} S_{1b} & = & \{seq_1, seq_2\} & \quad & S_{2b} & = & \{seq_4\} \\ S_{1r} & = & \{seq_3\} & & S_{2r} & = & \{seq_5\} \end{array}$$

We obtain for the function *insert*: $\Delta_{1b} = [zs \geq 1 \rightarrow [\rho_2 \mapsto 2]] \sqcup [zs \geq 2 \rightarrow [\rho_2 \mapsto 2]] = [zs \geq 1 \rightarrow [\rho_2 \mapsto 2]]$ and $\Delta_{1r} = [zs \geq 2 \rightarrow [\rho_2 \mapsto 1]]$. By considering the *nb* and *nr* functions of Example 17 we get:

$$\Delta_0^{insert} = [zs \geq 2 \rightarrow [\rho_2 \mapsto zs + 1]] \sqcup [zs \geq 1 \rightarrow [\rho_2 \mapsto 2]]$$

By attaching this abstract heap to *insert* in the signature environment, we obtain the following results for *insSort*: $\Delta_{2b} = [zs \geq 1 \rightarrow [\rho_2 \mapsto 1]]$ and $\Delta_{2r} = [zs \geq 3 \rightarrow [\rho_2 \mapsto zs]] \sqcup [zs \geq 2 \rightarrow [\rho_2 \mapsto 2]]$, so

$$\begin{array}{rcl} \Delta_0^{insSort} & = & [zs \geq 3 \rightarrow [\rho_2 \mapsto zs(zs-1)+1]] \sqcup \\ & & [zs \geq 2 \rightarrow [\rho_2 \mapsto 2(zs-1)+1]] \sqcup \\ & & [zs \geq 1 \rightarrow [\rho_2 \mapsto 1]] \end{array}$$

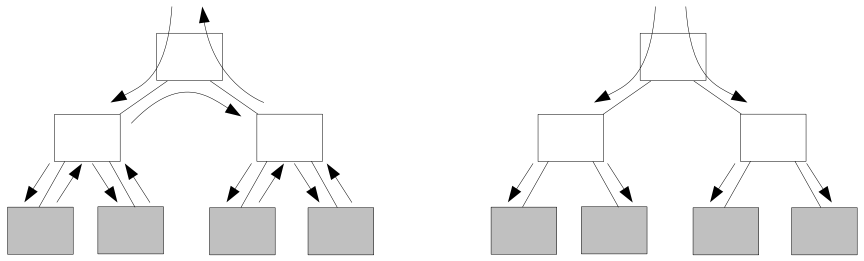
□

4.1.2. Algorithm for computing μ_0

The algorithm for computing a first approximation μ_0 to the heap needs of a function is more involved. Let us consider the call tree of a given function call f. We make a distinction between the charges done to the working regions (*self*) of the calls in this tree, and the charges done to the remaining regions. The latter are cumulative, in the sense that the cells created in these regions are not removed while the execution of the root call to f progresses. The arrows in Figure 17a represent the directions of the execution flow, in which charges to these regions grow. With respect to the charges done to the working regions of the calls of the activation tree, these only grow from the root call to the base cases, as Figure 17b shows. In this case we no longer have arrows pointing upwards in the tree, because all the cells created in the working region are removed when its corresponding function call finishes. Therefore, the only directions in which we know for sure that these charges grow, are the paths from the root call to its recursive children.

Now assume that, during this call to f, the execution flow has reached the point just before executing the last base call (see Figure 18). We assume the worst-case execution in which the longest call chain is the one who leads to the last base call. In this situation, we obtain:

- The maximum accumulation of charges done to the regions different from *self*. This is represented in Figure 18a. We denote these charges by Δ_{bef}, which can be obtained by applying the $[\![\cdot]\!]_\Delta$-interpretation to the execution sequences up to (and including) the last recursive call.



(a) Regions different from the working region

(b) Working region

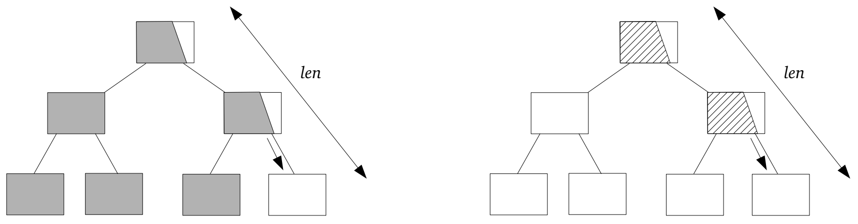
Figure 17: Growth of the charges done to different regions as the execution progresses.

- The maximum simultaneous charges done to the *self* region of the current sequence of calls. These charges are represented in Figure 18b. If we denote by Δ_{self} the charges done in this region by each individual call, and by *len* the length of the longest call chain, then the expression $\Delta_{self} * (len - 1)$ is an upper bound of these charges. As in the previous case, the value of Δ_{self} is obtained by applying the $[\![\cdot]\!]_\Delta$-interpretation without considering the expressions being executed after the last recursive call.

The combination of these two charges gives us $\Delta_{self} * (len - 1) + \|\Delta_{bef}\|$. Taking this value as a base level, we have to take the following charges into account:

- Maximum level of memory occupied before the execution flow reaches the last base call, on account of the charges done in memory before the first recursive call and between any two consecutive recursive calls (μ_{bef}).
- Memory needs of the last base case, which is going to be executed (μ_b). This corresponds to the white-marked call of Figure 18a.
- Memory needs of the part of the recursive cases which is still to be executed after the last recursive call (μ_{aft}). This corresponds to the white part of the gray-marked cells whose execution is pending.

Since none of them is necessarily greater than the other ones, we take the least upper bound of the three. The algorithm is shown in Figure 19. Notice that



(a) Regions different from *self* (b) Working region (*self*)

Figure 18: Execution point before the control flow reaches the last recursive call. Full gray nodes in (a) represent those calls whose execution is finished, whereas gray-white nodes represent those calls whose execution has taken place until the last recursive call. The rightmost base case (white node) has not been executed yet.

it must receive an abstract heap Δ, which is usually the initial approximation Δ_0 given by *computeDelta*, although any other sound approximation to the charges done by f is valid. We assume the existence of a function $\mathit{splitBA}_f$ which divides each execution sequence in S_r into two subsequences: one with the expressions being executed before (and including) the last recursive call, and another with the expressions being executed from then. These are stored, respectively, in S_{bef} and S_{aft}. The abstract interpretations are applied to these sets as explained previously. In the same way as in our Δ_0, we add μ_b in the least upper bound in order to deal with $\perp$ in the remaining components.

Example 19. Starting from the $\Delta_0^{\mathit{insert}}$ computed in Example 18, the algorithm *computeMu*, when applied to *insert*, yields the following partial results

$$\begin{array}{rcl}
\|\Delta_{\mathit{bef}}\| & = & [zs \geq 3 \rightarrow zs] \sqcup [zs \geq 2 \rightarrow 2] \\
\Delta_{\mathit{self}} & = & [zs \geq 2 \rightarrow 0] \\
\mu_{\mathit{bef}} & = & [zs \geq 2 \rightarrow 0] \\
\mu_{\mathit{aft}} & = & [zs \geq 2 \rightarrow 1] \\
\mu_b & = & [zs \geq 1 \rightarrow 2]
\end{array}$$

which lead to the following initial bound:

$$\mu_0^{\mathit{insert}} = [zs \geq 3 \rightarrow zs + 2] \sqcup [zs \geq 2 \rightarrow 4] \sqcup [zs \geq 1 \rightarrow 2]$$

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

Figure 19: Algorithm for computing μ_0.

By using the previous $\Delta_0^{\mathit{insert}}$, μ_0^{insert}, the algorithm applied to *insSort* yields the following

$$
\begin{array}{rcl}
\|\Delta_{\mathit{bef}}\| & = & [zs \geq 4 \to (zs-1)(zs-2)+1] \sqcup \\
& & [zs \geq 3 \to 2(zs-2)+1] \sqcup \\
& & [zs \geq 2 \to 1]
\end{array}
$$

$$
\begin{array}{rcl}
\Delta_{\mathit{self}} & = & [zs \geq 2 \to 0] \\
\mu_{\mathit{bef}} & = & [zs \geq 2 \to 0] \\
\mu_{\mathit{aft}} & = & [zs \geq 4 \to zs+1] \sqcup [zs \geq 3 \to 4] \sqcup [zs \geq 2 \to 2] \\
\mu_b & = & [zs \geq 1 \to 1]
\end{array}
$$

which results in the following initial bound:

$$
\begin{array}{rcl}
\mu_0^{\mathit{insSort}} & = & [zs \geq 4 \to zs^2 - 2zs + 4] \sqcup \\
& & [zs \geq 3 \to 2zs + 1] \sqcup \\
& & [zs \geq 2 \to 3] \sqcup \\
& & [zs \geq 1 \to 1]
\end{array}
$$

□

4.1.3. Algorithm for computing σ_0

In order to approximate the stack costs of a function, we follow an approach similar to that of μ_0. In this case we do not have cumulative components such as the Δ_{bef} shown before, because the behaviour of the stack, in this sense, is analogous to that of the *self* region in the heap: it grows as the execution flow descends through the activation tree. We use the term

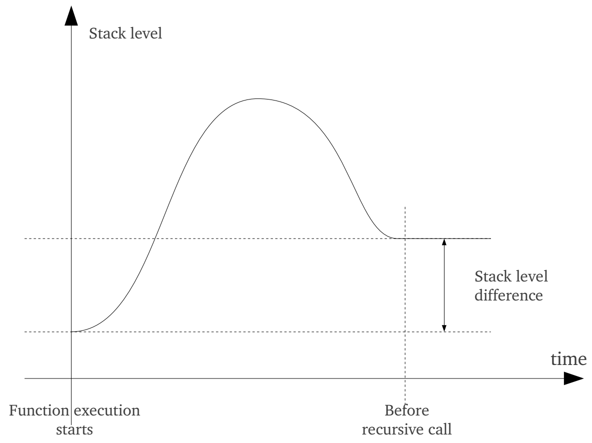


Figure 20: The stack level difference value denotes the maximum difference between the stack levels when the function starts executing and before its recursive call is done.

stack level to denote the number of words existing in the stack at a given execution point. It is useful to obtain the maximum difference between the stack levels in two execution points: when the context function f starts its execution, and when a recursive call to f is going to be done (see Figure 20). We use the term *stack level difference* to refer to this value, which is not expressed as an element of $\mathbb{F}$, but as an element of $\mathbb{N}^{\perp}$ instead. This is because this value does not depend on the input sizes as the rest of the components we have seen so far. As we will see below, this quantity can be statically approximated.

In order to compute an initial σ_0 let us replicate the behaviour shown in Figure 20 along a number *len* of nested recursive calls, so that we get the graph shown in Figure 21. At the point in which the last base case is about to be executed, the stack level reaches $SD * (len - 1)$ words, being SD the maximum stack level difference. Then we have to proceed similarly as in the computation of μ_0: taking the value $SD * (len - 1)$ as a base level, we should consider the stack costs on account of the base cases and the part of the recursive cases before and after the last recursive call. If we denoted these components by σ_b, σ_{bef} and σ_{aft} respectively, we would have to take the least upper bound $\sqcup\{\sigma_b, \sigma_{bef}, \sigma_{aft}\}$. However, and because the absence of cumulative components in the computation of stack needs (unlike the heap needs μ, in which the cumulative Δ component is involved), taking the least upper bound of these three components is equivalent to applying the $[\![\cdot]\!]_\sigma$-

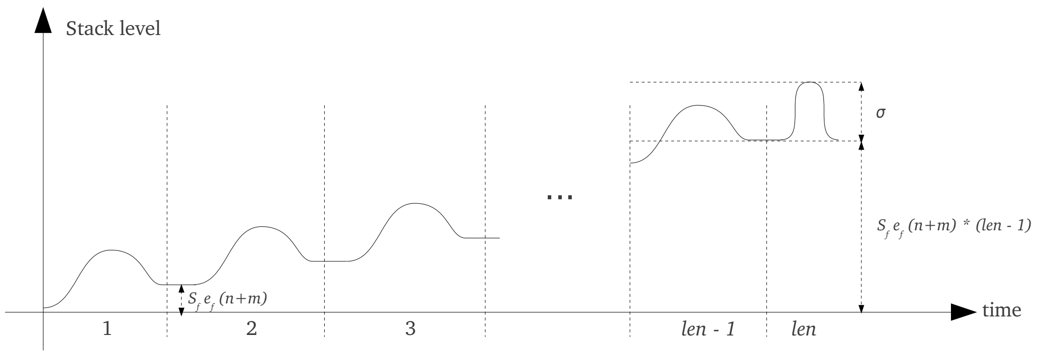


Figure 21: Stack level behaviour during the execution of subsequent recursive calls.

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

Figure 22: Computation of an initial approximation σ_0.

interpretation to the whole expression without taking into account the costs of the subsequent recursive calls.

The algorithm for computing σ_0 is shown in Figure 22. The *SD* function, defined in Figure 23, computes the stack level difference for a given expression in the context of a function definition. If this expression does not contain recursive calls, it returns $\bot$. If e_f is the body of the context function f, with n data parameters and m region parameters, the result of $SD_f\ e_f\ (n+m)$ is the stack level difference of this function.

Example 20. The algorithm *computeSigma*, when applied to the *insert* function, yields the following intermediate results

$$SD_{insert}\ (e_{insert})\ 3 = 8 \qquad \sigma = [zs \geq 2 \rightarrow 8] \sqcup [zs \geq 1 \rightarrow 3]$$

leading to $\sigma_0^{insert} = [zs \geq 2 \rightarrow 8zs] \sqcup [zs \geq 1 \rightarrow 8zs - 5]$. For the function *insSort* we obtain

$$SD_{insSort}\ (e_{insSort})\ 2 = 6$$
$$\sigma = [zs \geq 3 \rightarrow 8zs - 7] \sqcup [zs \geq 2 \rightarrow \sqcup\{6, 8zs - 12\}] \sqcup [zs \geq 1 \rightarrow 1]$$

$$\begin{array}{lcl}
SD_f\ (f\ \overline{a}\ @\ \overline{r})\ td & = & |\overline{a}| + |\overline{r}| - td \\
SD_f\ be\ td & = & \bot \qquad \text{if } be \text{ is not a call to } f \\
SD_f\ (\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2)\ td & = & \sqcup\{2 + SD_f\ e_1\ 0, 1 + SD_f\ e_2\ (td+1)\} \\
SD_f\ (\textbf{case}\ x\ \textbf{of}\ \overline{alt})\ td & = & \sqcup_{r=1}^{n}(|\overline{x_r}| + SD_f\ e_r\ (td + |\overline{x_r}|)) \\
 & & \textbf{where}\ alt_r = C_r\ \overline{x_r} \to e_r
\end{array}$$

Figure 23: Definition of SD_f, which computes the stack level difference.

leading to

$$\begin{array}{rcl}
\sigma_0^{insSort} & = & [zs \geq 3 \to 14zs - 13] \sqcup \\
 & & [zs \geq 2 \to \sqcup\{6zs, 14zs - 18\}] \sqcup \\
 & & [zs \geq 1 \to 6zs - 5]
\end{array}$$

□

4.2. Correctness of the initial approximations

4.2.1. Before/after semantics with call tree counters

In this section we make precise the idea of the *nb*, *nr*, and *len* functions being correct approximations to the actual parameters of the call tree deployed at runtime. This is done by extending our semantic judgements with a triple (N_b, N_r, L) of natural numbers representing the number of base calls, recursive calls, and the maximum depth of the call tree. Obviously, it makes little sense to talk about recursive calls, if we do not specify *which* function do these numbers refer to. Therefore, the name of this function will be attached to the arrow of these judgements, as in $\Downarrow_f$.

Besides this, it turns out to be useful to distinguish between the charges done in memory before the last recursive call in the current context, and those done after this call. For this reason, we shall split each of the δ and m components of our resource vector into two subcomponents: their *before* part, (δ_b and m_b), and their *after* part (δ_a and m_a). The former takes into account the expressions being executed before (and including) the last recursive call in the current call context, and the latter takes into account the expressions being executed after (and excluding) that recursive call. By convention, we assume that, if the evaluation of an expression does not contain any recursive call, all its charges are stored in the after part, and the before part is left with no charges.

Given the above, our extended semantics defines the derivation of judgements of the following form:

$$E \vdash h, k, e \Downarrow_f h', k, v, (N_b, N_r, L), (\delta_b/\delta_a, m_b/m_a)$$

We omit the s component of the resource vector, since no distinction between the before and after part is necessary for that component, as we shall see later. In Figure 24 we show the rules for **let** expressions. The rest can be found in [34, 35].

When evaluating a **let** expression we have to distinguish whether a recursive call to f is done during the evaluation of e_2. This can be done with the help of the L_2 component of the call tree counter returned by the evaluation of e_2. If there are no recursive calls in e_2 we obtain $L_2 = 1$ (i.e. the depth of the call tree is 1), and we apply $[Let1_{NC}]$. In this case, the before part corresponds to the before part of the evaluation of e_1, whereas the after part comprises the after part of e_1 and the whole evaluation of e_2. The $[Let2_{NC}]$ rule is used when there are recursive calls in e_2 (that is, $L_2 > 1$). The before part includes the evaluation of e_1 and the before part of e_2, whereas the after part only contains the after part of e_2. In this case, the formalization of the (N_r, N_b, L) requires further case distinction, and we define a separate operator $\otimes$ for this purpose.

From these semantic rules it is easy to show that the new components introduced are just counters, and they do not influence the evaluation of the expression. As a consequence, if we can execute an expression e under the $\Downarrow$ semantics for a given environment E and heap h with k regions, we can do the same under the $\Downarrow_f$ semantics (for any f), and we will get the same normal form v and final heap h'. A little less obvious is the relation between the $(\delta_b/\delta_a, m_b/m_a)$ components of the $\Downarrow_f$ semantics, and the original resource vector (δ, m, s) given by the corresponding $\Downarrow$-judgement. The whole δ component is equivalent to the (region-wise) addition of the before and after parts. With respect to the m component, an intuitive idea of its relation with the m_b and m_a was already given by Section 2.3, when the heap consumption of two expressions executed in sequence was explained. In this case, we can consider that the consumption done by an expression is equivalent to the sequential execution of the before and after parts of such expression. The following lemma states these results formally.

Lemma 1. *Let us assume an evaluation* $E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$. *Given a function* f, *there exists unique tuples* (N_b, N_r, L) *and* $(\delta_b/\delta_a, m_b/m_a)$

$$\frac{\begin{array}{c} L_2 = 1 \\ E \vdash h, k, e_1 \Downarrow_f h', k, v_1, (N_{b,1}, N_{r,1}, L_1), (\delta_{b,1}/\delta_{a,1}, m_{b,1}/m_{a,1}) \\ E \cup [x_1 \mapsto v_1] \vdash h', k, e_2 \Downarrow_f h'', k, v, (N_{b,2}, N_{r,2}, L_2), (\delta_{b,2}/\delta_{a,2}, m_{b,2}/m_{a,2}) \\ \delta_a = \delta_{a,1} + \delta_{a,2} \qquad m_a = \max\{m_{a,1}, \|\delta_{a,1}\| + m_{a,2}\} \end{array}}{E \vdash h, k, \textbf{let } x_1 = e_1 \textbf{ in } e_2 \Downarrow_f h'', k, v, (N_{b,1}, N_{r,1}, L_1), (\delta_{b,1}/\delta_a, m_{b,1}/m_a)} \; [\mathit{Let1}_{NC}]$$

$$\frac{\begin{array}{c} L_2 \neq 1 \\ (N_b, N_r, L) = (N_{b,1}, N_{r,1}, L_1) \otimes (N_{b,2}, N_{r,2}, L_2) \\ E \vdash h, k, e_1 \Downarrow_f h', k, v_1, (N_{b,1}, N_{r,1}, L_1), (\delta_{b,1}/\delta_{a,1}, m_{b,1}/m_{a,1}) \\ E \cup [x_1 \mapsto v_1] \vdash h', k, e_2 \Downarrow_f h'', k, v, (N_{b,2}, N_{r,2}, L_2), (\delta_{b,2}/\delta_{a,2}, m_{b,2}/m_{a,2}) \\ \delta_b = \delta_{b,1} + \delta_{a,1} + \delta_{b,2} \qquad m_b = \max\{m_{b,1}, \|\delta_{b,1}\| + m_{a,1}, \|\delta_{b,1}\| + \|\delta_{a,1}\| + m_{b,2}\} \end{array}}{E \vdash h, k, \textbf{let } x_1 = e_1 \textbf{ in } e_2 \Downarrow_f h'', k, v, (N_b, N_r, L), (\delta_b/\delta_{a,2}, m_b/m_{a,2})} \; [\mathit{Let2}_{NC}]$$

$$\begin{array}{l} (N_{b,1}, N_{r,1}, L_1) \otimes (N_{b,2}, N_{r,2}, L_2) = (N_b, N_r, \max\{L_1, L_2\}) \\ \textbf{where } (N_b, N_r) = \begin{cases} (N_{b,2}, N_{r,2}) & \text{if } N_{r,1} = 0 \\ (N_{b,1} + N_{b,2}, N_{r,1} + N_{r,2} - 1) & \text{otherwise} \end{cases} \end{array}$$

Figure 24: Enriched big-step semantics

such that $E \vdash h, k, e \Downarrow_f h', k, v, (N_b, N_r, L), (\delta_b/\delta_a, m_b/m_a)$. *Moreover, it holds that:*

1. $\delta = \delta_b + \delta_a$.
2. $m = \max\{m_b, \|\delta_b\| + m_a\}$.

Proof. The uniqueness of (N_b, N_r, L) and $(\delta_b/\delta_a, m_b/m_a)$ are consequence of the $\Downarrow$-evaluation being deterministic. The remaining properties are proven by induction on the size of the $\Downarrow$-derivation.

Similarly as we did in Section 3.3, we can prove that the abstract interpretation of the before and after sequences are, respectively, upper bounds of the corresponding $(\delta_b/\delta_a, m_b/m_a)$ components of the semantics.

In the following we shall use the letters φ, ψ, χ, etc. to denote $\Downarrow$-judgements. Given one of these:

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

we have shown that, given a function f, there exist unique numbers N_b, N_r, L, m_b, and m_a, and unique mappings δ_a, δ_b such that $E \vdash h, k, e \Downarrow_f h', k, v, (N_b, N_r, L), (\delta_b/\delta_a, m_b/m_a)$ can be derived. All these components are determined by the judgement φ, so we can use the notation $N_b^f(\varphi)$, $N_r^f(\varphi)$, $L^f(\varphi)$, $\delta_b^f(\varphi)$, and $\delta_a^f(\varphi)$ to refer to these components. By abuse of notation, we use $\delta(\varphi)$, $m(\varphi)$, $s(\varphi)$ for denoting, respectively, the δ, m, and s components of the resource vector occurring in the judgement φ. Moreover, we use $Exp(\varphi)$ for denoting the expression being evaluated in φ. The notation $\Phi(\varphi)$ denotes the set of judgements belonging to the derivation of φ (including φ itself).

Now we are ready to give a formal definition of a function *nb*, *nr* and *len* being correct.

Definition 9. Let us assume a function definition $f\ \overline{x}\ @\ \overline{r} = e_f$. We say that *nr* (resp. *nb* and *len*) is a correct approximation of the number of recursive calls (resp. base calls and height of the call tree), iff for every φ, E_f, h, k, td, h', v, δ, m, s, $\overline{v}$, $\bar{\imath}$, $\overline{s}$ such that

1. $\varphi \equiv E_f \vdash h, k, td, e_f \Downarrow h', k, v, (\delta, m, s)$, where $E_f = [\overline{x} \mapsto \overline{v}, \overline{r} \mapsto \bar{\imath}, \mathit{self} \mapsto k+1]$.
2. For each $i \in \{1..|\overline{x}|\}$, $s_i = \mathit{size}(h, v_i)$

then it holds that $nr\ \overline{s} \geq N_r^f(\varphi)$ (resp. $nb\ \overline{s} \geq N_b^f(\varphi)$ and $len\ \overline{s} \geq L^f(\varphi)$).

4.2.2. Correctness of the initial Δ_0, μ_0 and σ_0

The first step for proving the correctness of *computeDelta* is to express our δ component resulting from the evaluation of a given function in terms of the charges done by the base and recursive cases, and the number of base and recursive calls done during that evaluation. The intuition behind this idea is the same as in Section 4.1.1. In order to compute the charges done by a recursive call, we need to be able to isolate the charges done by the call itself from the charges done by its subsequent recursive calls. This can be done with the help of an additional syntactic construct **dmask**, which resets the δ component of a given expression. Its semantics are given by the following rule:

$$\frac{E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)}{E \vdash h, k, td, \mathbf{dmask}\ e \Downarrow h', k, v, ([\,], m, s)}$$

It is easy to see that if we replace a function definition $f\ \overline{x}\ @\ \overline{r} = e_f$ in a environment **FD** by another masked function definition $f\ \overline{x}\ @\ \overline{r} = \mathbf{dmask}\, e_f$

we will be able to obtain the same judgements as with our initial signature, but we will obtain a possibly different δ component in the resource vector. The next theorem uses this function to define the charges done by a recursive call.

Theorem 2. *Assume a function definition* $f\ \overline{x}\ @\ \overline{r} = e_f \in \mathbf{FD}$ *such that the following execution takes place:*

$$\varphi \equiv E \vdash h, k+1, td, e_f \Downarrow_{\mathbf{FD}} h', k+1, v, (\delta, m, s)$$

Let us define $\mathbf{FD}' = (\mathbf{FD} \backslash f) \uplus [f \mapsto f\ \overline{x}\ @\ \overline{r} = \mathbf{dmask}\ e_f]$, *and assume the following execution under* $\mathbf{FD}'$,

$$\varphi' \equiv E \vdash h, k+1, td, e_f \Downarrow_{\mathbf{FD}'} h', k+1, v, (\delta', m, s)$$

which is derivable by using the $\Downarrow$*-rules. Given the following definitions*

$$\begin{array}{rcl} \delta_{base} & = & \max\{\delta(\psi) \mid \psi \in \Phi(\varphi), Exp(\psi) = e_f, L^f(\psi) = 1\} \\ \delta_{rec} & = & \max\{\delta(\psi) \mid \psi \in \Phi(\varphi'), Exp(\psi) = e_f, L^f(\psi) > 1\} \end{array}$$

then we get:

$$\delta|_k \leq \delta_{base}|_k * N_b^f(\varphi) + \delta_{rec}|_k * N_r^f(\varphi) \tag{6}$$

Proof. By induction on $L^f(\varphi)$.

Notice the similarity between the expression (6) and that occurring in the definition of *computeDelta*. The latter can be considered an "abstract" version of the former. The correctness of the result of *computeDelta* follows from this theorem.

Theorem 3. *Let* $\Delta_0 = computeDelta\ (f\ \overline{x}\ @\ \overline{r} = e_f)\ \Sigma\ \Gamma\ nb\ nr$. *If the following conditions hold*

1. Σ *contains correct signatures for all the functions being called from* f, *except* f *itself.*
2. *The expression* e_f *is typeable under the typing environment* Γ.
3. *nb and nr are correct approximations of the number of base and recursive calls of* f, *respectively.*
4. *The abstract heaps* Δ_b *and* Δ_r *occurring in the definition of computeDelta are parameter-decreasing.*

then Δ_0 *is a correct abstract heap for* f*.*

Proof. (Sketch) It is a consequence of Theorems 1 and 2. Assume a judgement:

$$\varphi \equiv E \vdash h, k, td, e_f \Downarrow h', k, v, (\delta, m, s)$$

By Theorem 1, the abstract heap Δ_b is a correct bound of $\delta_{base}|_k$, and Δ_r is a correct bound for δ_{rec}, because $[\,]$ is a correct bound to the heap charges of the function definition $f\ \overline{x}\ @\ \overline{r} = \mathbf{dmask}\ e_f$.

Let us define, for each $i \in \{1..|\overline{x}|\}$, $s_i = \mathit{size}(h, E(x_i))$, and let us denote by η the region instantiation consistent with E and Γ. We get:

$$\Delta_0 \sqsupseteq \lfloor \Delta_b \rfloor * nb + \lfloor \Delta_r \rfloor * nr \succeq_{\overline{s},k,\eta} \delta_{base}|_k * N_b^f(\varphi) + \delta_{rec}|_k * N_r^f(\varphi) \geq \delta|_k$$

In order to prove the correctness of *computeMu* we use an additional construct **mmask**, which resets the m component of the resource vector:

$$\frac{E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)}{E \vdash h, k, td, \mathbf{mmask}\ e \Downarrow h', k, v, (\delta, 0, s)}$$

Similarly as before, the following theorem gives the memory needs as an expression which resembles the cost expression given by *computeMu*. The difference is that it uses elements from the concrete domain, rather than the abstract one.

Theorem 4. *Assume a function definition* $f\ \overline{x}\ @\ \overline{r} = e_f \in \mathbf{FD}$ *and the following judgement:*

$$\varphi \equiv E \vdash h, k+1, td, e_f \Downarrow_{\mathbf{FD}} h', k+1, v, (\delta, m, s)$$

Let us define $\mathbf{FD}' = (\mathbf{FD} \backslash f) \uplus [f \mapsto f\ \overline{x}\ @\ \overline{r} = \mathbf{mmask}\ e_f]$*, and assume the following execution under* $\mathbf{FD}'$*,*

$$\varphi' \equiv E \vdash h, k+1, td, e_f \Downarrow_{\mathbf{FD}'} h', k+1, v, (\delta, m', s)$$

which is derivable by using the $\Downarrow$*-rules. Given the following definitions:*

$$\begin{array}{rcl}
\delta_{self} & = & \max\{\delta_b^f(\psi)(k+1) \mid \psi \in \Phi(\varphi), \mathit{Exp}(\psi) = e_f, L^f(\psi) > 1\} \\
m_{bef} & = & \max\{m_b^f(\psi) \mid \psi \in \Phi(\varphi'), \mathit{Exp}(\psi) = e_f, L^f(\psi) > 1\} \\
m_{aft} & = & \max\{m_a^f(\psi) \mid \psi \in \Phi(\varphi), \mathit{Exp}(\psi) = e_f, L^f(\psi) > 1\} \\
m_{base} & = & \max\{m(\psi) \mid \psi \in \Phi(\varphi), \mathit{Exp}(\psi) = e_f, L^f(\psi) = 1\}
\end{array}$$

We get:

$$m \leq \|\delta_b^f(\varphi)|_k\| + \delta_{self} * (L^f(\varphi) - 1) + \max\{m_{bef}, m_{aft}, m_{base}\} \qquad (7)$$

Proof. By induction on $L^f(\varphi)$, using Lemma 1.

The correctness of *computeMu* is established by connecting the elements of the abstract domain appearing in its result with the elements of the concrete domain occurring in the previous theorem.

Theorem 5. *Let* $\mu_0 = \mathit{computeMu}\ (f\ \overline{x}\ @\ \overline{r} = e_f)\ \Sigma\ \Gamma\ \Delta\ \mathit{len}$. *If the following conditions hold*

1. Σ *contains correct signatures for all the functions being called from* f*, except* f *itself.*
2. *The expression* e_f *is typeable under the typing environment* Γ.
3. *len is a correct approximation of the maximum length of the call tree of* f.
4. Δ *is a correct abstract heap for* f.
5. *The space cost functions* Δ_{self}, Δ_{bef}, μ_{aft}, μ_{bef}, *and* μ_b *occurring in the definition of computeMu are parameter-decreasing.*

then μ_0 *is correct with respect to* f.

Proof. (Sketch) The proof proceeds in a similar way as Theorem 3. In this case, it follows from Theorems 1 and 4. Assume a judgement

$$\varphi \equiv E \vdash h, k, td, e_f \Downarrow h', k, v, (\delta, m, s)$$

and denote, for each $i \in \{1..|\overline{x}|\}$, $s_i = \mathit{size}(h, E(x_i))$,

By Theorem 1, and following the same steps as in proof of Theorem 3 we can prove that $\mu_b \succeq_{\overline{s}} m_{\mathit{base}}$.

Similarly, we can prove that $\Delta_{\mathit{self}} \succeq_{\overline{s}} \delta_{\mathit{self}}$, $\|\Delta_{\mathit{bef}}\| \succeq_{\overline{s}} \|\delta_b^f(\varphi)|_k\|$, $\mu_{\mathit{bef}} \succeq_{\overline{s}} m_{\mathit{bef}}$, $\mu_{\mathit{aft}} \succeq_{\overline{s}} m_{\mathit{aft}}$. So we obtain:

$$\begin{array}{rcl} \mu_0 & \sqsupseteq & \Delta_{\mathit{self}} * (\mathit{len} - 1) + \|\Delta_{\mathit{bef}}\| + \sqcup\{\mu_{\mathit{bef}}, \mu_{\mathit{aft}}, \mu_b\} \\ & \succeq_{\overline{s}} & \delta_{\mathit{self}} * (L^f(\varphi) - 1) + \|\delta_b^f(\varphi)|_k\| + \max\{m_{\mathit{bef}}, m_{\mathit{aft}}, m_{\mathit{base}}\} \\ & \geq & m \end{array}$$

For proving the correctness of *computeSigma* we follow a similar approach. The details can be found in [35].

4.3. Correctness in absence of parameter-decrease conditions.

The correctness of the initial bounds depend on the fact that their components are parameter-decreasing, as stated in Definition 8. Therefore, it is useful to establish some sufficient conditions under which the parameter-decrease property is guaranteed. Given a function definition, the first condition states that if the sizes of the parameters do not increase from the root call to the recursive ones, every abstract heap or space cost function is parameter-decreasing with respect to that definition.

Proposition 1. *Assume a function definition* $f\ \overline{x}\ @\ \overline{r} = e_f$*. If for every recursive call* $f\ \overline{a}\ @\ \overline{r}$ *in its body it holds that* $|a_k|\ \overline{x} \sqsubseteq x_k$ *for every* $k \in \{1..|\overline{x}|\}$*, then every* $\Delta \in \mathbb{D}$ *and* $\xi \in \mathbb{F}$ *is parameter-decreasing with respect to* f*.*

It is not unusual to write function definitions for which this condition does not hold. In particular, those functions having an accumulator parameter whose size increases from the root call to its recursive calls do not satisfy this condition. Notice, however, that in these kind of functions, the costs do not depend on the sizes of these accumulator parameters. Hence we can set out the following weaker sufficient condition.

Proposition 2. *Assume a function definition* $f\ \overline{x}\ @\ \overline{r} = e_f$ *and* $\Delta \in \mathbb{D}$ *(resp.* $\xi \in \mathbb{F}$*). If for every recursive call* $f\ \overline{a}\ @\ \overline{r}$ *in the body of* f *it holds that* $|a_k|\ \overline{x} \sqsubseteq x_k$ *for every* k *of* $\{1..|\overline{x}|\}$ *except a subset* P*, and* Δ *(resp.* ξ*), does not depend on the parameters contained within* P*, then* Δ *(resp.* ξ*) is parameter-decreasing with respect to* f*.*

This criterion is satisfied by every example function definition in this paper. In absence of the parameter-decrease property, we can still adapt our *computeDelta*, *computeMu* and *computeSigma* algorithms so as to get correct initial approximations. We can follow an approach similar to that of [2]: the key idea is to compute an invariant Ψ bounding the feasible sizes of the parameters as a function of the arguments given to the root call, and then maximize the partial components appearing in each algorithm (for instance, Δ_b and Δ_r in *computeDelta*). If the invariant Ψ is given by a set of linear constraints, we can use linear programming techniques for this maximization. This adaptation is subject of future work.

5. Conclusions and related work

We have introduced an abstract interpretation-based analysis for computing memory bounds, which takes heap and stack consumption into account. In this paper we show how to calculate initial correct approximations to the consumption of recursive functions. It is possible to apply repeatedly the abstract interpretation to the initial approximations and obtain also correct approximations. In [34] we show that, under certain conditions, the new approximations are more accurate than the previous ones.

The strengths of our approach can be summarized as follows:

1. It scales well to large programs, as each *Safe* function can be separately inferred. The relevant information about the called functions is recorded in the signature environment.
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.

Safe's type system [29] 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. The restriction we have imposed to our cost and size functions to be non-negative and monotone is the reason why destructive pattern matching has been omitted from our analysis in a first phase. In [34] we provide some sufficient conditions in which cell deallocation can be taken into account without breaking the monotonicity restriction. Polymorphic recursion does not fit either in our inference strategy for Δ_f, since it assumes that the regions used by a recursive function f do not change from the external call to the internal ones. As the number of region combinations is finite, we could extend our algorithms to polymorphic recursion with some additional analysis on these region patterns, as we explain in [34].

The first approaches to space consumption analysis were restricted to infer linear memory bounds. Hughes and Pareto developed in [24] a type system and a type-checking algorithm which guarantees safe memory upper bounds in a region-based first-order functional language. Unfortunately, the approach requires the programmer to provide detailed consumption annotations. As it has been said, the first fully automatic technique is due to Hofmann and Jost. In [21] they present a type system and a type inference algorithm which, in case of success, guarantees linear heap upper bounds for a first-order functional language, and it does not require programmer annotations. This type system has been extended in [27] so as to support higher-order functions, and in [43] to support lazy evaluation.

Beyond linear bounds, the pioneer research on memory consumption is that carried out under the AHA project (*Amortised analysis of Heap space Usage*) [16], aimed at inferring amortised costs for heap space. In [40], Shkaravska et al. introduce a variant of sized types, in which the size annotations can be polynomials of any degree. They address two novel problems: polynomials are not necessarily monotonic and they are *exact* bounds, as opposed to approximate upper bounds. These bounds are inferred with a combination of testing and polynomial interpolation-based techniques. In [39] they extend their work to give approximate upper bounds on the output sizes, thus broadening the class of analysable programs. In this case, the size relations are expressed via non-deterministic conditional rewriting systems, from which a closed form is extracted by using polynomial interpolation. A strength of this approach is that, since the inference is testing-based, the function being inferred can be considered as a black box. This allows them to use the same inference techniques in different applications, such as, for example, inferring loop-bounds for Java programs [41]. Unfortunately, polynomial interpolation-based techniques do not necessarily lead to a correct upper bound, and some external mechanism is needed for checking that the result of the analysis is sound. Our analysis always gives correct bounds if the externally given call-tree and size information is correct.

The COSTA system (*COSt and Termination Analyzer for Java Bytecode*) [3, 4] implements a fully mechanical approach to resource analysis for Java bytecode programs. It is based on the classical method of Wegbreit [47]. It consists in the generation of a recurrence relation which captures the cost of the program being analysed, and the computation of a closed form by using a built-in recurrence solver PUBS [1, 2]. Their results go far beyond linear bounds: the system can infer polynomial, logarithmic, and exponential bounds. COSTA also allows a restricted form of non-monotonicity, provided it occurs in the context of linear expressions. The computation of our initial Δ_0 is inspired by the way in which PUBS solves recurrence relations. The main difference is that PUBS computes the *nb* and *nr* functions giving the number of calls in a recurrence, whereas these functions are given externally in our system. However, PUBS' approach of computing both *nr* and *nb* from the *len* function may yield imprecise over-approximations in some cases (e.g. Quicksort). A drawback, in comparison to our system, is that COSTA does not support non-linear size relations, even if these were given externally. The reason behind this is that these relations are the guards of the recurrence being generated, and PUBS assumes these guards to be conjunctions of linear constraints.

More recently, the COSTA team has extended their system in order to deal with different models of garbage collection [5]. The new results are very promising. In their work, they claim that their liveness-based model can accommodate the region-based memory management approach of [10], although this integration is neither described nor formally specified.

Another promising technique for inferring polynomial bounds is due to Hoffmann and Hofmann [20], which extends the work of [21]. In the univariate case, their system is able to infer bounds, expressed as non-negative linear combinations of binomial coefficients. These combinations subsume the class of polynomials with non-negative coefficients, while allowing some polynomials with negative coefficients, such as $x^2 - x$. However, it does not cover some other natural-valued polynomial bounds, such as $3x^2 - 6x + 7$. In a more recent work [19], Hoffmann et al. extend their analysis to the inference of multivariate functions. Unlike our system, they do not handle regions nor explicit destruction, although they claim that the latter can be added with no difficulty. Their first attempts only dealt with lists but recent papers can also deal with arbitrary data types.

Both the original type system of [21], and its extension to polynomial bounds [20] are closely related to the potential method used in the context of amortised analysis [45, 13]. This approach provides an implicit notion of *input size* (the number of elements in a list, usually), but there is no explicit dependence between input sizes and costs, since the latter are given by the potential assigned to each element of the input and output DSs. On the contrary, our space analysis, as well as the COSTA system, are both based on explicit sizes and symbolic manipulation. It is arguable whether one is better than the other. An advantage of amortised analysis is that it allows to express more precisely the memory costs, when they do not depend exclusively on the input size. Moreover, amortised analysis yields more precise results when considering several functions executed in sequence, since it accounts for the overall costs as a whole, instead of just adding the worst-case costs of each function separately. On the contrary, the approaches based on symbolic manipulation can be extended more easily, for instance, by including cost expressions from several complexity classes, as it is done in *Safe*.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, Automatic inference of upper bounds for recurrence relations in cost analysis, in: S.B.. Heidelberg (Ed.), Static Analysis: 15th International Symposium, SAS'08, LNCS 5079, Springer, 2008, pp. 221–237.

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

[3] 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.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Costa: Design and implementation of a cost and termination analyzer for java bytecode, in: Formal Methods for Components and Objects, FMCO'08, LNCS 5382, Springer, 2008, pp. 113–132.

[5] 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.

[6] 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.

[7] F. Benoy, A. King, Inferring argument size relationships with CLP($\mathcal{R}$), in: Proceedings of the 6th International Workshop on Logic Programming Synthesis and Transformation, LOPSTR'96, LNCS 1207, Springer, 1997, pp. 204–223.

[8] B. Campbell, Prediction of linear memory usage for first-order functional programs, in: Trends in Functional Programming. Volume 9. Selected papers of the 9th Symposium on Trends in Functional Programming, TFP'08, Intellect, 2008, pp. 1–16.

[9] B. Campbell, Type-based amortised stack memory prediction, Ph.D. thesis, Laboratory for Foundations of Computer Science. School of Informatics. University of Edinburgh, 2008.

[10] S. Cherem, R. Rugina, Region analysis and transformation for java programs, in: Proceedings of the 4th International Symposium on Memory Management, ISMM'04, ACM, 2004, pp. 85–96.

[11] W.N. Chin, S.C. Khoo, Calculating sized types, Higher Order and Symbolic Computation 14 (2001) 261–300.

[12] J. Conesa, R. López, A. Lozano, Desarrollo de un compilador para un lenguaje funcional con gestión explícita de memoria, Master's thesis, Universidad Complutense de Madrid, 2006.

[13] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to algorithms, Introduction to Algorithms, MIT Press, 2009, third edition, pp. 451–463.

[14] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'77, ACM Press, 1977, pp. 238–252.

[15] J. de Dios, R. Peña, Certification of Safe Polynomial Memory Bounds, in: Int. Symp. on Formal Methods, FM'11, Limerick, Ireland, LNCS 6664, Springer, 2011, pp. 184–199.

[16] M.v. Eekelen, O. Shkaravska, R.v. Kesteren, B. Jacobs, E. Poll, S. Smetsers, AHA: Amortized space usage analysis, in: Trends in Functional

Programming. Volume 8. Selected Papers of the 7th Symposium on Trends in Functional Programming, TFP'07, Intellect, 2008, pp. 36–53.

[17] C. Flanagan, A. Sabry, B.F. Duba, M. Felleisen, The essence of compiling with continuations, in: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, PLDI'93, ACM, 1993, pp. 237–247.

[18] G. Grov, G. Michaelson, C. Herrmann, H.W. Loidl, S. Jost, K. Hammond, Hume cost analyses for imperative programs, in: Proceedings of International Conference on Software Engineering Theory and Practice, SETP'09, ISRST, 2009, pp. 16–23.

[19] 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.

[20] 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.

[21] 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.

[22] M. Hofmann, S. Jost, Type-based amortised heap-space analysis, in: Proceedings of the 15th European Symposium on Programming, ESOP'06, LNCS 2924, Springer, 2006, pp. 22–37.

[23] P. Hudak, J. Hughes, S.P. Jones, P. Wadler, A history of Haskell: being lazy with class, in: Proceedings of the third ACM SIGPLAN conference on History of programming languages, HOPL III, ACM, 2007, pp. (12–1)–(12–55).

[24] 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.

[25] 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.

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

[27] 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.

[28] S. Lucas, R. Peña, Rewriting techniques for analysing termination and complexity bounds of Safe programs, in: Draft Proceedings of 18th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR'08, Universidad Politécnica de Valencia, 2008, pp. 43–57.

[29] 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.

[30] 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.

[31] M. Montenegro, R. Peña, C. Segura, A resource-aware semantics and abstract machine for a functional language with explicit deallocation, in: M. Falaschi (Ed.), 17th International Workshop on Functional and Logic Programming, WFLP'08, Electronic Notes in Computer Science, vol. 246, 2009, pp. 167–182.

[32] 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.

[33] 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 (2014) 33 pages. To appear. Available online.

[34] 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.

[35] M. Montenegro, R. Peña, C. Segura, Space Consumption Analysis by Abstract Interpretation. Inference of Recursive Functions. Detailed Proofs, 2014. Pages 1–41.

[36] M. Montenegro, O. Shkaravska, M. van Eekelen, R. Peña, Interpolation-based height analysis for improving a recurrence solver, in: Proceedings of the 2nd International Workshop on Foundational Practical Aspects of Resource Analysis, FOPARA'11, LNCS 7177, Springer, 2012, pp. 36–53.

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

[38] R. Peña, A.D. Delgado, Size invariant and ranking function synthesis in a functional language, in: Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming, WFLP'11, LNCS 6816, Springer, 2011, pp. 52–67.

[39] O. Shkaravska, M. van Eekelen, A. Tamalet, Collected size semantics for functional programs over lists, in: Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL'08, LNCS 5836, Springer, 2008, pp. 1–21.

[40] O. Shkaravska, M.v. Eekelen, R.v. Kesteren, Polynomial size analysis of first-order shapely functions, Logical Methods in Computer Science 5 (2009) 1–35. Special Issue with Selected Papers from TLCA'07.

[41] O. Shkaravska, R. Kersten, M. van Eekelen, Test-based inference of polynomial loop-bound functions, in: Int. Conf. on the Principles and Practice of Programming in Java, PPPJ'10, ACM, 2010, pp. 99–108.

[42] F. Siebert, Hard Realtime Garbage Collection in Modern Object Oriented Programming, Books on Demand GmbH, 2002.

[43] 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.

[44] F. Spoto, P.M. Hill, É. Payet, Path-length analysis for object-oriented programs, in: EAAI'06: First International Workshop on Emerging Applications of Abstract Interpretation, AAAI Press, 2006.

[45] R.E. Tarjan, Amortized computational complexity, SIAM Journal on Algebraic and Discrete Methods 6 (1985) 306–318.

[46] P.B. Vasconcelos, K. Hammond, Inferring cost equations for recursive, polymorphic and higher-order functional programs, in: Int. Work. on Impl. of Funct. Lang., IFL'03, LNCS 3145, Springer, 2004, pp. 86–101.

[47] B. Wegbreit, Mechanical program analysis, Communications of the ACM 18 (1975) 528–539.