A Resource Semantics and Abstract Machine for *Safe*

A Functional Language with Regions and Explicit Deallocation[☆]

Manuel Montenegro, Ricardo Peña, Clara Segura

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Abstract

In this paper we summarize *Safe*, a first-order functional language for programming small devices and embedded systems with strict memory requirements, which has been introduced elsewhere. It has some unusual memory management features such as heap regions and explicit cell deallocation. It is targeted at a Proof Carrying Code environment, and consistently with this aim the *Safe* compiler provides machine checkable certificates about important safety properties such as absence of dangling pointers and bounded memory consumption.

The kernel of the paper is devoted to developing part of the *Safe* compiler's back-end, by deriving an appropriate abstract machine from the language semantics, by providing the code generation functions, and by formally proving that the translation is sound, both in the semantic and in the memory consumption senses.

Keywords: functional languages, memory management, certifying compilers, abstract machines, code generation

1. Introduction

The first-order functional language *Safe* has been developed in the last few years as a research platform for analysing and formally certifying proper-

[☆]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)

ties of programs related to memory usage. It was introduced for investigating the suitability of functional languages for programming small devices and embedded systems with strict memory requirements. One of its aims is to infer and certify —at compile time— safe upper bounds on memory consumption by following a Proof Carrying Code (PCC) approach [1]. The *Safe* compiler produces as target language Java bytecode, so that *Safe* programs can be executed in most mobile devices and web browsers.

Several features make *Safe* different from conventional functional languages:

1. A *region based* memory management system, so that a garbage collector is not needed.
2. A programmer facility for explicit destruction of memory cells. These could be immediately reused by the program, so reducing its memory requirements.
3. The type system of the language guarantees that well-typed programs will be free of dangling pointers at runtime.

The compiler also includes an abstract interpretation-based inference algorithm computing upper bounds to the heap and stack consumed by a program. The latter two properties (absence of dangling pointers and memory bounds) are certified by the compiler.

In most functional languages, memory management is delegated to the runtime system: fresh heap memory is allocated during program evaluation as long as there is enough free memory available. When there is not, the garbage collector interrupts program execution in order to copy or mark the live part of the heap, and the remaining memory is considered to be free. The main advantage of this approach is that programmers do not bother about low level memory management details.

But there are also some drawbacks: the time delay introduced by garbage collection may prevent the program from providing an answer in a required reaction time, memory exhaustion in safety critical systems may provoke an unacceptable damage to users, and it is difficult to predict at compile time the data structure lifetimes, and to reason about memory consumption.

There has been some work on on-the-fly garbage collectors which try to compact memory during the idle time of the time-critical threads, but this approach does not guarantee either to recover enough memory in the available time, or to meet the real-time constraints. In [2] it is claimed to guarantee

both things at the same time by calling the memory recovery operations *within* the critical threads, and by guaranteeing a worst-case execution time for these operations. The penalty appears to be having a rather complex system.

As said above, *Safe* is targeted to programming small devices, safety critical and soft real time systems, where memory requirements are rather strict, and naive garbage collectors are a burden in service availability. The constant time *Safe* memory management primitives, and the above certified properties, make *Safe* suitable for these purposes.

Much work has been done on *Safe* up to now:

- A battery of static analyses has been developed and implemented in the *Safe* compiler. In [3, 4, 5] we describe different parts of the type system and of the type inference algorithms which, as a whole, determine the regions, allocate data structures into them, and guarantee that the destruction facilities do not create dangling pointers.

- A further analysis is presented in [6] for inferring safe memory bounds. For each *Safe* function, these bounds consist of multivariate symbolic functions relating the sizes of its input arguments to the number of cells consumed in the heap, and of words consumed in the stack.

- In [7, 8] we explain how to certify the above properties inferred by the compiler, i.e. the absence of dangling pointers, and the correctness of the memory consumption upper bounds [1].

- Finally, in [9, 10] we certify the translation to the JVM target code. The above analyses and certificates are carried on at the compiler intermediate language called *Core-Safe*, which is still functional. The compiler's back-end translates *Core-Safe* to JVM bytecode in two phases: from *Core-Safe* to the *Safe*'s virtual machine (SVM), and from there to JVM. In a PCC framework, it is mandatory to certify that the properties holding at the *Core-Safe* level still hold at the JVM level.

This paper is an extended version of the work described in [11] and additionally it is intended as a reference of the *Safe* project. For that reason

[1] At `http://dalila.sip.ucm.es/safe/certifdangling` and `.../safe/bounds` the reader can find the Isabelle/HOL theories related to these certificates.

it includes an extensive introduction to *Safe*'s features and their motivation. It also includes some implementation aspects of the *Safe* runtime system. The focus of the paper is on the first phase of *Safe*'s compiler back-end, i.e. the design of the *Safe Virtual Machine* and the translation algorithms from *Core-Safe* to the SVM language. Then, we formally prove that the semantics are preserved across the translation. Originally, the design of the SVM was done by a derivation consisting of three succesive refinements: from *Core-Safe*'s big-step operational semantics to a small-step semantics, from there to an intermediate machine called M2, and from this one to the SVM. We omit the intermediate steps here which the interested reader can find in [11].

An original contribution of [11] (and hence of this extended version) is that we also formally prove the memory consumption correctness. To this aim, we previously enrich both the big-step semantics and the SVM semantics with additional information expressing the memory consumption at each of the levels. The correctness proof contained in the paper is hand-written, but a more involved one including all the details was done by using the proof-assistant Isabelle/HOL [12] and published in [9]. It is remarkable that the actual Haskell code of the *Safe* compiler performing the translation from *Core-Safe* to SVM has been extracted from the Isabelle/HOL code on which the formal proof was performed. The same strategy was followed in the translation from SVM to JVM.

More information about the *Safe* project, including the publications and also the Isabelle/HOL theories used for certification, is available at `http://dalila.sip.ucm.es/safe`. There also exists a web-based interface for the compiler at `http://dalila.sip.ucm.es/~safe`.

Firstly, by means of examples we present in Section 2 a high-level view of *Safe* features, and how its memory facilities are implemented. Then, its desugared variant *Core-Safe* is defined. Its semantics is given in Section 3. In Section 4 we present the SVM abstract machine on which *Safe* programs are run, while the translation process between *Core-Safe* programs and the code being executed by the SVM abstract machine is detailed in Section 5. The SVM will serve as a reference implementation, which allows us to determine the memory consumption of a *Core-Safe* program. This will be made apparent in Section 6 by enriching the *Core-Safe* semantics with a resource vector specifying how much memory is needed by a program in order to be run. A formal proof of correctness of this translation, showing both semantic and resource preservation, is done in Section 7. Finally, Section 8 surveys related work and concludes.

2. 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. As an example, we have the following **data** declarations of binary search trees and lists:

$$\begin{array}{lcl} \textbf{data}\ \mathit{BSTree}\ \alpha & = & \mathit{Empty} \mid \mathit{Node}\ (\mathit{BSTree}\ \alpha)\ \alpha\ (\mathit{BSTree}\ \alpha) \\ \textbf{data}\ [\alpha] & = & [\,] \mid (\alpha : [\alpha]) \end{array}$$

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* 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 we would have a fixed number of cell sizes, all of them multiples of the smallest one.

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 $C_1 \rightarrow C_2$, 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. 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 constitutes a separate DS on its own.

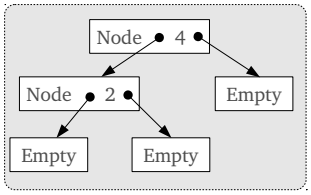
During the design of the language several decisions were taken. The first one involves the correspondence between DSs and regions.

Axiom 1. *A DS completely resides in a single region.*

This decision poses a constraint on the data constructors: the recursive children of a cell (i.e. those with the same type) must belong to the same region as the father.

Axiom 2. *A DS can be part of another DS, and two DSs may share a third DS.*

As an example, consider the binary tree on the right-hand side. The left and the right subtrees of the root are separate DSs, which belong to the whole binary tree, which is another DS.



Axiom 3. *Basic values (integers and booleans) occurring in the heap do not belong to any region by themselves. They are contained within cells.*

2.1. Region-based memory management in Safe

A distinctive aspect of *Safe* is the way in which regions are created and destroyed:

Axiom 4. *Allocation of regions takes place at function calls. Deallocation of regions takes place when a function call finishes.*

This implies that 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, regions also have nested lifetimes. That is why they are stored in a stack-like fashion: the last region being created is the first being destroyed. The region associated to a given function call f is called its *working region*. The function may create DSs in this region, provided these DS 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 stack bottom region) 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 the *Safe* programmer. The compiler determines which DSs will be created in the working region and which regions should be passed as parameters between functions. 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 region parameters will contain, at runtime, the identifiers (natural numbers) of the actual regions they refer to.

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

- When calling a function with these extra parameters, the region arguments are also separated from the rest of arguments by the @ symbol. For example:

$$f\ 4\ x\ z\ @\ self\ r_1$$

  where r_1 is a region variable in scope.

- To each constructor expression, a region variable is attached which contains, at runtime, the identifier of the region where the resulting cell will be built. For example:

$$[\,]\ @\ r_2 \qquad (4\ :\ [\,]\ @\ self\ )\ @\ self$$

  In the latter example, the outermost *self* annotates the application of the list constructor (:).

Example 5. Consider a function *append* for appending two lists. The following is *Safe* code, as written by the programmer:

$$\begin{array}{lllll} append & :: & [\alpha] \to [\alpha] \to [\alpha] & & \\ append & [\,] & ys & = & ys \\ append & (x : xs) & ys & = & x : append\ xs\ ys \end{array}$$

This function is annotated by the compiler as follows:

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

There is a new region parameter r, which is used to build the resulting list, and is passed to the subsequent recursive calls. Figure 1 shows the execution

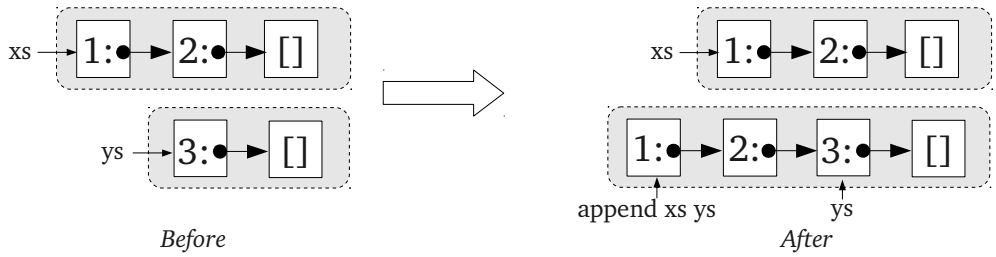


Figure 1: Regions of the arguments before and after evaluating *append xs ys*. The result of the function application is created in the region of *ys*.

of *append xs ys*, being $xs = [1, 2]$ and $ys = [3]$. Notice that the result is forced to be built in the same region as the list passed as second parameter. This is because this parameter is reused in the base case of *append*, and a DS must be contained within a single region. The compiler takes this into account when annotating the call to *append*. □

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*.

Example 6. Consider the following implementation of the *Treesort* algorithm:

$$\begin{aligned} &treesort \ :: \ [\alpha] \to [\alpha] \\ &treesort \ xs = inorder \ (mkTree \ xs) \end{aligned}$$

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. The *inorder* function receives the parameter given to *treesort*, which is the output region in which the sorted list will be built (Figure 2). When *treesort* finishes, its working region will disappear from the heap, together with the temporary tree. □

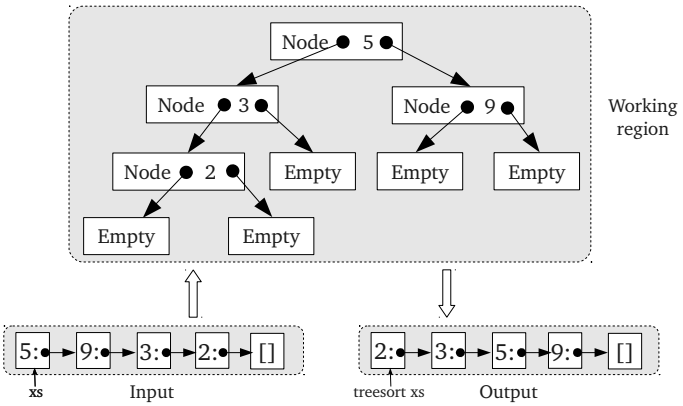


Figure 2: DSs involved in the *treesort* function. The working region contains the intermediate representation of the list as a binary tree.

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 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 7. The *append* function of Example 5 forces the resulting list to be located in the same region as the list passed as second parameter. 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:

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

The copy of ys is created in the output region r, which may now be different from the region of the second parameter ys. (Figure 3). □

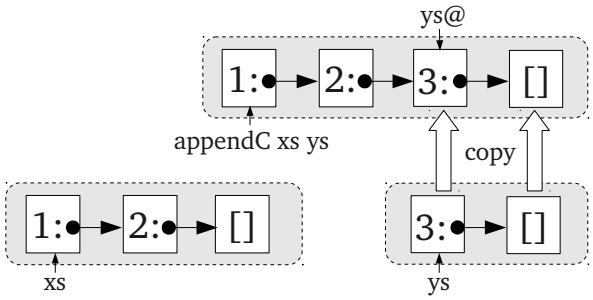


Figure 3: Runtime behaviour of *appendC* [1, 2] [3]: the list passed as second parameter is copied, so it does not share cells with the result, which may be built in an independent region.

Example 8. The copy of a DS might not be able to live in a region different from that of the DS being copied. In the following example,

$$duplicate\ t = Node\ t\ 0\ (t@)$$

the original binary tree t and its copy $t@$ are forced to live in the same region, as they belong to the same DS. □

2.2. Destructive pattern matching

Destructive pattern matching allows the selective disposal of a DS inside a region, without the need of waiting for the whole region to be disposed of. This allows the programmer to break the strict discipline imposed by the nested lifetimes of regions.

Destructive pattern matching, denoted by (!) or a **case**! expression, deallocates the cell corresponding to the outermost constructor of the DS being matched against. In this case we say that the DS involved in the destructive pattern matching is *condemned*.

Axiom 9. *A function may only read a DS which is not a condemned parameter, and it may read (before destroying) and destroy a DS which is a condemned parameter.*

As an example, we will consider a destructive variant of *append*.

Example 10. Assume the following definition:

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

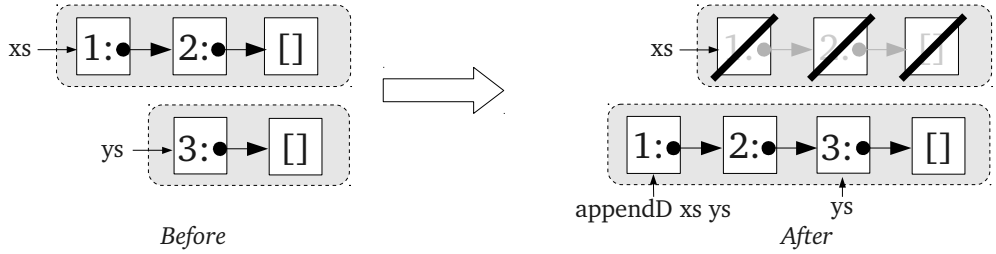


Figure 4: Regions of the input parameters before and after a call to *appendD* [1, 2] [3]. Every cell of the input list is removed in each recursive call to *appendD*.

The (!) mark in the first parameter specifies that the cell to which the pattern matching is done will be destroyed at runtime. The function destroys the first cons cell of the list passed as first parameter. The remaining cells will be destroyed in the subsequent recursive calls to *appendD*, until we reach the base case in which the empty list matches the first equation, and it is also destroyed (Figure 4). This version needs no additional heap space: a cell is destroyed and another cell is built in each recursive call. The destruction of the first parameter is reflected in the function's type: $appendD :: [\alpha]! \rightarrow [\alpha] \rightarrow [\alpha]$. □

Destructive pattern matching allows the programmer to define functions requiring constant additional heap space. As we have said before, it is also useful for breaking the restriction (imposed by regions) of having DSs with nested lifetimes.

Example 11. Given a list of elements, the *insertion sort* algorithm starts with an empty list and does successive ordered insertions in it with the elements of the input list. The *insert* function does the insertion of an element in an ordered list. Here we show its *Full-Safe* code and the corresponding region-annotated version:

$$
\begin{array}{ll}
insert\ x\ [\,] = [x] & insert\ x\ [\,]\ @\ r = (x : [\,]\ @\ r)\ @\ r \\
insert\ x\ (y : ys) & insert\ x\ (y : ys)\ @\ r \\
\quad \mid x \leq y = x : y : ys & \quad \mid x \leq y = (x : (y : ys)@r)@r \\
\quad \mid x > y = y : insert\ x\ ys & \quad \mid x > y = (y : insert\ x\ ys\ @\ r)@r
\end{array}
$$

In the equation guarded by $x \leq y$ the result is built upon the input ys, so the region of the output list must be the same as that of the input list

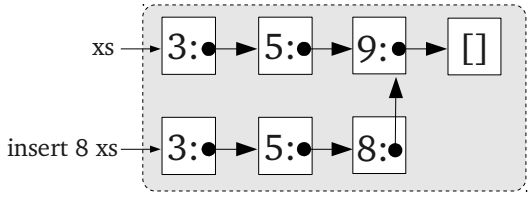


Figure 5: Inserting the number 8 in the list $[3, 5, 9]$. The part of the list before the new cell must be reconstructed.

(Figure 5). As a result, the following *inssort* function,

$$\begin{array}{ll} \mathit{inssort}\ [\,] = [\,] & \mathit{inssort}\ [\,]\ @\ r = [\,]\ @\ r \\ \mathit{inssort}\ (x : xs) = & \mathit{inssort}\ (x : xs)\ @\ r = \\ \qquad \mathit{insert}\ x\ (\mathit{inssort}\ xs) & \qquad \mathit{insert}\ x\ (\mathit{inssort}\ xs\ @\ r)\ @\ r \end{array}$$

builds every intermediate result in the region of the empty list being created in the base case, that is, the output region r, whereas the working regions of the calls to *inssort* remain unused. This implies that the *inssort* function has a $\mathcal{O}(n^2)$ worst-case space complexity, where n is the number of elements of the input list.

A more efficient approach is to consider a destructive version of *insert*:

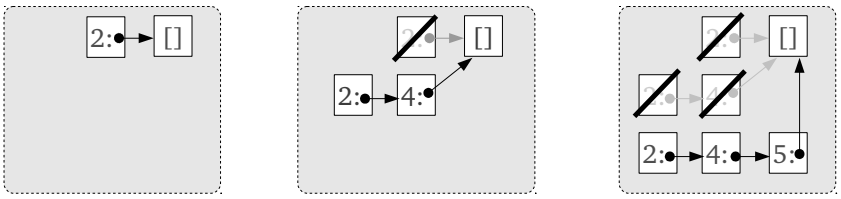
$$\begin{array}{l} \mathit{insertD}\ x\ [\,]! = [x] \\ \mathit{insertD}\ x\ (y : ys)! \\ \qquad \mid x \leq y = x : y : ys \\ \qquad \mid x > y = y : \mathit{insertD}\ x\ ys \end{array}$$

This version only needs an additional cell in memory to build the new list node. Assuming that we replace the call to *insert* by *insertD* in the *inssort* function, Figure 6 shows the state of the output region when returning from every recursive call. We can even develop a destructive version of *inssort* that also disposes of the input list.

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

The cell of the input list being destroyed in the pattern matching can be reused by the *insertD* function. Thus *inssortD* needs no additional heap space for building the result.

□



(a) *inssort* [2] finishes (b) *inssort* [4,2] finishes (c) *inssort* [5,4,2] finishes

Figure 6: Insertion sort destroying the intermediate results.

2.3. Core-Safe syntax

The functions presented in previous sections were written in *Full-Safe*, which is the language in which the programmer writes his programs. *Core-Safe* is a desugared version of *Full-Safe* in which regions are explicit.

In Figure 7 we show the syntax of *Core-Safe* programs and expressions. A program *prog* is a sequence of **data** declarations, followed by a sequence of function definitions $\overline{def}_i$ and a main expression e, whose result is the result of the program. The **data** declarations section follows a syntax similar to that of Haskell and it is not described here. A function definition is a function name f, followed by a list $\overline{x_i}$ of formal parameters (which are variables), a list $\overline{r_j}$ of formal *region* parameters (which are called *region variables*) and the *body* expression e of the function. The sets of variables and region variables are respectively denoted by **Var** and **RegVar**.

We denote by **Exp** the set of *Core-Safe* expressions. Basic expressions include: atomic expressions (literals or variables), copy expressions (see Section 2.1), function and constructor applications, and a special kind of function applications that we consider to be built-in: basic operators. 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 that, for every constructor C, the set of its recursive positions (denoted by $RecPos(C)$) is known at runtime. For instance, $RecPos(:) = \{2\}$ and $RecPos(Node) = \{1, 3\}$. Only atomic expressions (constants and variables) are allowed in function and constructor applications. Nonatomic expressions occurring inside function and constructor arguments must be introduced via **let** bindings, in the style of A-normal form [13].

Core-Safe supports two kinds of pattern matching: read-only (**case**) and destructive (**case**!). In the latter, the cell against which the patterns are

$$
\begin{array}{rcll}
\mathit{prog} & \to & \overline{\mathit{data}_i};\ \overline{\mathit{def}_i}\ ;e & \\
\mathit{def} & \to & f\ \overline{x_i}\ @\ \overline{r_j} = e & \\
e & \to & a & \{\text{atom: literal } c \text{ or variable } x\} \\
 & & \mid x\ @\ r & \{\text{copy}\} \\
 & & \mid a_1 \oplus a_2 & \{\text{basic operator application}\} \\
 & & \mid C\ \overline{a_i}\ @\ r & \{\text{constructor application}\} \\
 & & \mid f\ \overline{a_i}\ @\ \overline{r_j} & \{\text{function application}\} \\
 & & \mid \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 & \{\textbf{let} \text{ declaration: nonrecursive, monomorphic}\} \\
 & & \mid \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i} & \{\text{read-only pattern matching}\} \\
 & & \mid \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i} & \{\text{destructive pattern matching}\}
\end{array}
$$

Figure 7: *Core-Safe* language definition

matched is also disposed of, so its space can be reused by the runtime system. When translating from *Full-Safe* into *Core-Safe*, pattern matching and destructive pattern matching are respectively translated into **case** and **case!** expressions.

Example 12. The translation applied to the *append* function defined in Example 5 yields the following result:

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

□

2.4. Static semantics

A part of the *Safe*'s compiler front-end is devoted to ensuring certain static properties that are assumed to be true in the dynamic semantics of Section 3.

The first one is that all bound variables are distinct within a function definition, and even (although this is not needed by the semantics) between different function definitions. This is ensured by systematically renaming them during the translation from *Full-Safe* to *Core-Safe*.

Another one is that pattern matching is translated in such a way that **case** expressions are exhaustive, i.e. they have an alternative for each constructor

$$\frac{\Gamma_1 \vdash e_1 : s_1 \qquad \Gamma_2 + [x_1 : \tau_1] \vdash e_2 : s \qquad \mathit{utype?}(\tau_1, s_1) \quad \forall x \in \operatorname{dom}\Gamma_1.\Gamma_1(x) \in \mathbf{UnsafeType} \Rightarrow x \notin \mathit{fv}(e_2)}{\Gamma_1 \sqcup \Gamma_2 \vdash \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 : s}\ [\text{LET}]$$

$$\frac{\forall i \in \{1..n\}.\Gamma(C_i) = \sigma_i \quad \mathit{utype?}(\Gamma(x), T\ @\ \rho) \quad \forall i \in \{1..n\}.\overline{s_{ij}}^{n_i} \to \rho \to T\ @\ \rho \trianglelefteq \sigma_i \quad \forall i \in \{1..n\}.\Gamma + [\overline{x_{ij} : \tau_{ij}}^{n_i}] \vdash e_i : s \quad \forall i \in \{1..n\}.\forall j \in \{1..n_i\}.\mathit{utype?}(s_{ij}, \tau_{ij})}{\Gamma \vdash \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n} : s}\ [\text{CASE}]$$

Figure 8: Two typing rules of *Safe*'s type system

of the corresponding data type. If the programmer has chosen not to match a particular constructor (as in e.g. `head (x:xs) = x`), the compiler introduces the missing branch (in this example, `head []`) with an error expression.

The last one is that all running programs are well-typed, and then the semantics cannot get stuck because of a type error. For instance, if a **case** expression has a discriminant x of type T, then the value of x in the heap will correspond to this type, and consequently it will match one of the **case** patterns.

The *Safe* type system (see [3, 4, 5] for a full description) is an extension of the polymorphic Hindley-Milner (H-M) type system used in many functional languages. It has two additional features:

- Region arguments are given polymorphic type variables as types, reflecting the fact that these region arguments may be instantiated at runtime to any actual region. The only restriction imposed by the type system is that region arguments having the same type are instantianted to the same actual region.

- Algebraic datatypes are enriched with *destruction marks* indicating the safety degree of the corresponding data structure. These are *safe* (s), *condemned* (d), and *in-danger* (r) marks respectively. There is a total order between them, $s \leq d \leq r$, and a mark may be weakened if needed to a worse one. These marks are used to control the effect of **case**! expressions, avoiding that the program may access to partially or completely destroyed structures.

We have developed inference algorithms both for regions and for destruction marks so that this type system is essentially hidden to the programmer. In Figure 8 we show two of its rules.

In the [LET] rule, the predicate $utype?(\tau_1, s_1)$ expresses that the H-M types (i.e. the types without considering marks) of τ_1 and s_1 should be the same. The set **UnsafeType** contains the types with marks d or r, and $\Gamma_1 \sqcup \Gamma_2$ denotes the union of two typing environments such that the types of the common variables are weakened to the worse mark of the two. The rule essentially expresses that the variables becoming unsafe in expression e_1 may not be referenced in e_2.

In the [CASE] rule, the σ_i denote polymorphic type schemes, $t \trianglelefteq \sigma$ denotes that the type t is an instantiation of σ, and ρ is the region type variable assigned to the algebraic type T. The rule expresses that variables x having any mark can occur in the discriminant position of a **case**, and that its H-M type must be consistent with that of the constructors occurring in the patterns.

We show below the types inferred for some of the functions presented up to now:

$$
\begin{array}{ll}
\mathit{append} & :: [\alpha]@\rho_1 \to [\alpha]@\rho_2 \to \rho_2 \to [\alpha]@\rho_2 \\
\mathit{treesort} & :: [\alpha]@\rho_1 \to \rho_2 \to [\alpha]@\rho_2 \\
\mathit{appendC} & :: [\alpha]@\rho_1 \to [\alpha]@\rho_2 \to \rho_3 \to [\alpha]@\rho_3 \\
\mathit{duplicate} & :: \mathit{BSTree}\ \alpha\ @\ \rho \to \rho \to \mathit{BSTree}\ \alpha\ @\ \rho \\
\mathit{inssortD} & :: [\alpha]!@\rho_1 \to \rho_2 \to [\alpha]@\rho_2
\end{array}
$$

The (!) after $[\alpha]$ in the latter example is the mark d, indicating that the spine of the input list will be destroyed by *inssortD*.

2.5. Runtime system implementation

As we said above, the heap is implemented as a stack of regions. Each region is pushed initially empty, this action being associated to a *Safe* function invocation. During function execution new cells can be added to (or removed from) any active region as a consequence of constructor applications and destructive pattern matching. Upon function termination the whole topmost region is deallocated. The *Memory Management System* (MMS) maintains a pool of fresh cells, so that 'allocating' and 'deallocating' a cell respectively mean removing it from, or adding it to the pool. The MMS operations can be implemented in constant time by representing the regions and the pool as circular doubly-linked lists (see Figure 9). Removing a region amounts to joining two circular lists, which can obviously be done in constant time. The region stack is represented by a static array of dynamic lists, so that constant time access to each region is provided.

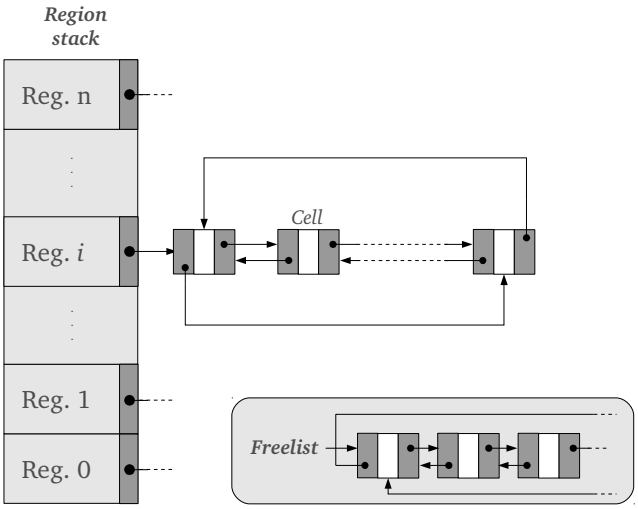


Figure 9: A picture of the *Safe* Virtual Machine heap and fresh cells pool

3. Semantics of *Safe*

In this section we describe how a *Core-Safe* expression e is reduced to a *value* (normal form). We use $v, v_i, \ldots$ metavariables to denote *values*, which are defined by the following grammar:

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

Since one of the aims of the language is the inference of safety properties regarding memory pointers and bounded memory consumption, our semantics needs a model of the heap. 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 $p, p_i, q, \ldots$ to denote elements from this set. A construction cell w is an element of the form $(j, C\ \overline{v_i}^n)$, where j is a natural number, C a constructor symbol of arity n, and $\overline{v_i}^n$ is the 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*).

The notation $region(w)$ represents the region where w lives (that is, the first component of the pair $(j, C\ \overline{v_i}^n)$), whereas $fresh_h(p)$ denotes that the pointer p is fresh in h, that is, it does not occur neither in its domain nor in its cells.

In Figure 10 we show the big-step operational semantics of *Core-Safe* expressions. A judgement of the form $E \vdash (h, k), e \Downarrow (h', k), v$ means that expression e is successfully reduced to a normal form v under a runtime environment E and a heap h with $k + 1$ regions (ranging from 0 to k) and that a final heap h' with the same number of regions as the initial one is produced as a side effect. A runtime environment E (also denoted *value environment*) maps respectively program variables x to values, and region variables r to heap region identifiers (i.e. natural numbers). We use $E[x \mapsto v]$ to highlight a binding contained in E, and $E \uplus [x \mapsto v]$ to denote the addition of a new binding to environment E, which requires $x \notin \text{dom } E$ in order to be defined. The same notation is applied to heaps. We adopt the convention that, for every value environment E, if c is a literal, $E(c) = c$.

We assume that, during the evaluation of an expression, a program signature Σ mapping function names to program definitions, is available. This is an additional parameter of the evaluation relation which is left implicit except for the rule $[App]$. We shall use the notation $(f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f) \in \Sigma$ for denoting the result of $\Sigma(f)$.

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

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

Now, we explain in detail the semantic rules. Rules $[Lit]$ and $[Var]$ just say that literals and heap pointers are normal forms. Rule $[Copy]$ executes a copy expression by copying 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. Some restricted type information is assumed to be available in our runtime system (namely, the recursive positions of each constructor) so that this function can be implemented.

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

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

$$\frac{}{E \vdash (h,k), a_1 \oplus a_2 \Downarrow (h',k), E(a_1) \oplus E(a_2)}\ [PrimOp]$$

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

$$\frac{E \leq k \qquad fresh_h(p)}{E \vdash (h,k), C\ \overline{a_i}^n \mathbin{@} r \Downarrow (h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)], k), p}\ [Cons]$$

$$\frac{(g\ \overline{y_i}^n \mathbin{@} \overline{r'_j}^m = e_g) \in \Sigma \qquad [\overline{y_i \mapsto E(a_i)}^n, \overline{r'_j \mapsto E(r_j)}^m, self \mapsto k+1] \vdash (h,k+1), e_g \Downarrow (h',k+1), v}{E \vdash (h,k), g\ \overline{a_i}^n \mathbin{@} \overline{r_j}^m \Downarrow (h'|_k, k), v}\ [App]$$

$$\frac{E \vdash (h,k), e_1 \Downarrow (h_1,k), v_1 \qquad E \uplus [x_1 \mapsto v_1] \vdash (h_1,k), e_2 \Downarrow (h',k), v}{E \vdash (h,k), \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 \Downarrow (h',k), v}\ [Let]$$

$$\frac{E \uplus [\overline{x_{ri} \mapsto v_i}^{n_r}] \vdash (h,k), e_r \Downarrow (h',k), v}{E[x \mapsto p] \vdash (h[p \mapsto (j, C_r\ \overline{v_i}^{n_r})], k), \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \Downarrow (h',k), v}\ [Case]$$

$$\frac{E \uplus [\overline{x_{ri} \mapsto v_i}^{n_r}] \vdash (h,k), e_r \Downarrow (h',k), v}{E[x \mapsto p] \vdash (h \uplus [p \mapsto (j, C_r\ \overline{v_i}^{n_r})], k), \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \Downarrow (h',k), v}\ [Case!]$$

Figure 10: Big-step operational semantics of *Core-Safe* expressions.

Definition 13. The *copy* function is defined as follows:

$$\begin{array}{rcl} \mathit{copy}(h_0[p \mapsto (j', C\ \overline{v_i}^n)], p, j) & = & (h_n \uplus [p' \mapsto (j, C\ \overline{v_i'}^n)], p') \\ \quad \textbf{where} \quad \mathit{fresh}_{h_n}(p') & & \\ \qquad \forall i \in \{1..n\}.(h_i, v_i') = \begin{cases} (h_{i-1}, v_i) & \text{if } i \notin \mathit{RecPos}(C) \\ \mathit{copy}(h_{i-1}, v_i, j) & \text{otherwise} \end{cases} & & \end{array}$$

Termination of this function is guaranteed because it is not possible to build cyclic data structures in *Safe*. We have proved this invariant property in [7]. Should *copy* find a dangling pointer during the traversal, then the whole rule would fail. If there is no failure, the normal form becomes 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. For instance, if the original DS is a list of lists, the structure created by *copy* is a copy of the outermost list, while the innermost lists become shared between the old and the new list.

Rule [*App*] shows when a new region is allocated. Notice that the body of the function 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 is a source of possible dangling pointers. By the notation $h|_k$ we denote the heap obtained by deleting from h those bindings living in regions greater than k.

Rule [*Cons*] generates a fresh location p pointing to the newly constructed cell. The parameters of the corresponding constructor are looked up in the value environment E.

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.

The [*Case*] rule is the usual one for an eager language, whereas the [*Case!*] rule expresses what happens in a destructive pattern matching: the binding of the discriminant variable disappears from the heap. This action is another source of possible dangling pointers.

In principle we are interested only in those value environments that map the *self* identifier to the highest possible region number, and the remaining

region variables to numbers strictly lower than the one bound to *self*. When a value environment meets these requirements, it is said to be *admissible*.

Definition 14. A value environment E is admissible with respect to k iff $E(\mathit{self}) = k$ and for every other region variable $r \in \operatorname{dom} E$ it holds that $E(r) < k$.

The next Proposition shows that admissibility is preserved by the evaluation of an expression.

Proposition 15. *Let us consider a judgement $E \vdash (h, k), e \Downarrow (h', k), v$ in which E is admissible w.r.t. k. Then any value environment occurring in the derivation of this judgement is also admissible w.r.t. its corresponding k.*

Proof. The property is true at the initial judgement and is preserved in every inductive rule. The only relevant case is rule $[App]$. □

It is easy to show that the initial value environment in the execution of a *Core-Safe* program (1) is admissible and, hence, that all the value environments taking place in the execution of the program are admissible. This allows us to leave out the conditions $j \leq k$ in $[Copy]$ and $E(r) \leq k$ in $[Cons]$, since they are guaranteed to hold when their corresponding judgements are admissible.

4. The *Safe* Virtual Machine (SVM)

In this section we formally describe an abstract machine for executing *Safe* expressions: the SVM (*Safe* Virtual Machine). This is an imperative machine: the expression being evaluated is replaced by a control sequence of imperative instructions. Correspondingly, the global code environment Σ is replaced by a *code store* containing sequences of instructions. Instead of runtime environments, the machine has a stack for storing values and continuations. The set of instructions and instruction sequences of the SVM are given by the following grammar:

$$
\begin{array}{rcl}
\iota & ::= & \texttt{DECREGION} \mid \texttt{COPY} \mid \texttt{MATCH}\ l\ \overline{\mathbf{p}_i} \mid \texttt{BUILDCLS}\ C\ \overline{K_i}\ K \\
& \mid & \texttt{POPCONT} \mid \texttt{CALL}\ \mathbf{p} \mid \texttt{MATCH!}\ l\ \overline{\mathbf{p}_i} \mid \texttt{SLIDE}\ m\ n \\
& \mid & \texttt{PUSHCONT}\ \mathbf{p} \mid \texttt{PRIMOP}\ \oplus \mid \texttt{BUILDENV}\ \overline{K_i} \\
\mathit{is} & ::= & [\,] \\
& \mid & \iota : \mathit{is}
\end{array}
$$

In this section we introduce only the semantics of these instructions in the context of the SVM machine. Their specific role will become clearer when dealing with the translation from *Core-Safe* to SVM code. During the execution the code store (resulting from the compilation of program fragments) is kept in the machine configuration. A code store maps *code pointers* to instruction sequences. We use the metavariables $\mathbf{p}, \mathbf{p}_i, ...$ for denoting code pointers. We assume the existence of a denumerable set **PCode** containing them. The l variable occurring in the `MATCH` and `MATCH!` instructions stands for a natural number representing a stack position (starting from the top). Finally, the elements K occurring in `BUILDENV` and `BUILDCLS`, which will be called *keys*, are generated by the following grammar:

$$\begin{array}{rcll} K & ::= & \texttt{Pos}\ j & \{\ j \in \mathbb{N}\text{, stack position }\} \\ & | & \texttt{Lit}\ c & \{\ \text{literal }\} \\ & | & \mathit{self} & \{\ \text{working region identifier }\} \end{array}$$

An SVM configuration c consists of six components $(is,\ h,\ k_0,\ k,\ S,\ cs)$, where is is the instruction sequence currently being executed, and h is a heap with k regions. The role of k_0 is more subtle, since it represents a lower watermark indicating which was the topmost region when the execution of the latest **let** started. This value is kept before executing the bound expression of that **let**. When this expression reaches a normal form, all the regions between the topmost k at that time, and the original topmost k_0 must be deleted, since the main expression of the **let** is going to be executed afterwards, and the operational semantic rules demand the initial and final regions to be equal when executing every expression. As the machine starts the execution of the bound expressions of successive nested **let**s, it has to store (with `PUSHCONT`) the k_0 corresponding to each of these expressions in the stack, and restore those intermediate values (with `POPCONT`) from the stack when the execution of their corresponding bound expressions finishes.

The cs variable denotes a code store containing all the program fragments, whereas S denotes a stack which may contain values, region numbers and continuations of the form $(k_0, \mathbf{p})$. We use the metavariable b for denoting stack values:

$$\begin{array}{rcll} b & ::= & j & \{\ \text{where } j \in \mathbb{N}\} \\ & | & v & \{\ \text{where } v \in \mathbf{Val}\} \\ & | & (k_0, \mathbf{p}) & \{\ \text{continuation: } k_0 \in \mathbb{N},\ \mathbf{p} \in \mathbf{PCode}\} \end{array}$$

Initial/final configuration	**Condition**
$(\texttt{DECREGION} : is,\ \ h,\ \ k_0,\ \ k,\ \ S,\ cs)$ $\rightarrow\ (is,\ \ h|_{k_0},\ \ k_0,\ \ k_0,\ \ S,\ cs)$	$k \geq k_0$
$([\texttt{POPCONT}],\ h,\ k,\ k,\ b : (k_0, \mathbf{p}) : S,\ cs[\mathbf{p} \mapsto is])$ $\rightarrow\ (is,\ \ h,\ \ k_0,\ \ k,\ \ b : S,\ \ cs)$	
$(\texttt{PUSHCONT}\ \mathbf{p} : is,\ \ h,\ \ k_0,\ \ k,\ \ S,\ cs)$ $\rightarrow\ (is,\ \ h,\ \ k,\ \ k,\ \ (k_0, \mathbf{p}) : S,\ \ cs)$	$\mathbf{p} \in \mathrm{dom}\ cs$
$(\texttt{COPY} : is,\ h,\ k_0,\ k,\ p : j : S,\ cs)$ $\rightarrow\ (is,\ \ h',\ \ k_0,\ \ k,\ \ p' : S,\ \ cs)$	$(h', p') = copy(h, p, j)$ $j \leq k$
$([\texttt{CALL}\ \mathbf{p}],\ \ h,\ \ k_0,\ \ k,\ \ S,\ \ cs[\mathbf{p} \mapsto is])$ $\rightarrow\ (is,\ \ h,\ \ k_0,\ \ k+1,\ \ S,\ \ cs)$	
$(\texttt{PRIMOP}\ \oplus : is,\ \ h,\ \ k_0,\ \ k,\ \ c_1 : c_2 : S,\ \ cs)$ $\rightarrow\ (is,\ \ h,\ \ k_0,\ \ k,\ \ c : S,\ \ cs)$	$c = c_1 \oplus c_2$
$([\texttt{MATCH}\ l\ \overline{\mathbf{p}_j}^m],\ h[S!l \mapsto (j, C_r^m\ \overline{v_i}^n)],\ k_0,\ k,\ S,\ cs[\overline{\mathbf{p}_j \mapsto is_j}^m])$ $\rightarrow\ (is_r,\ \ h,\ \ k_0,\ \ k,\ \ \overline{v_i}^n : S,\ \ cs)$	
$([\texttt{MATCH!}\ l\ \overline{\mathbf{p}_j}^m],\ h \uplus [S!l \mapsto (j, C_r^m\ \overline{v_i}^n)],\ k_0,\ k,\ S,\ cs[\overline{\mathbf{p}_j \mapsto is_j}^m])$ $\rightarrow\ (is_r,\ \ h,\ \ k_0,\ \ k,\ \ \overline{v_i}^n : S,\ \ cs)$	
$(\texttt{BUILDENV}\ \overline{K_i}^n : is,\ \ h,\ \ k_0,\ \ k,\ \ S,\ \ cs)$ $\rightarrow\ (is,\ \ h,\ \ k_0,\ \ k,\ \ \overline{Item_k(K_i)}^n : S,\ \ cs)$	(1)
$(\texttt{BUILDCLS}\ C_r^m\ \overline{K_i}^n\ K : is,\ \ h,\ \ k_0,\ \ k,\ \ S,\ \ cs)$ $\rightarrow\ (is,\ \ h \uplus [p \mapsto (Item_k(K), C_r^m\ \overline{Item_k(K_i)}^n)],\ \ k_0,\ \ k,\ \ p : S,\ \ cs)$	$Item_k(K) \leq k,\ fresh_h(p)$ (1)
$(\texttt{SLIDE}\ m\ n : is,\ \ h,\ \ k_0,\ \ k,\ \ \overline{v_i}^m : \overline{b_i}^n : S,\ \ cs)$ $\rightarrow\ (is,\ \ h,\ \ k_0,\ \ k,\ \ \overline{v_i}^m : S,\ \ cs)$	

(1) $Item_k(K) \stackrel{\text{def}}{=} \begin{cases} S!j & \text{if } K = \texttt{Pos}\ j \\ c & \text{if } K = \texttt{Lit}\ c \\ k & \text{if } K = self \end{cases}$

Figure 11: Transition rules of the SVM abstract machine

In Figure 11 we show the semantics of SVM instructions in terms of transitions between configurations. By C_r^m we denote the data constructor which is the r-th in its **data** definition out of a total of m data constructors. By $S!j$ we denote the j-th element of the stack S counting from the top and starting at 0 (i.e. $S!0$ is the topmost element). The notation $\overline{b_i}^n : S$ abbreviates the stack $b_1 : \cdots : b_n : S$.

Instruction `DECREGION` deletes from the heap all the regions, if any, between the current region k and region k_0, excluding the latter. It will be used when a normal form is reached.

Instruction `POPCONT` pops a continuation from the stack or stops the execution if there is none. It is always assumed to be the last element of the current instruction sequence (the notation [`POPCONT`] stands for `POPCONT` : []). Notice that b (which is usually a value) is left in the stack so that it can be accessed by the continuation. Instruction `PUSHCONT` pushes a continuation onto the stack. It is used in the translation of a **let**.

The `COPY` instruction just mimics the [*Copy*] rule of the big-step operational semantics, while leaving the result at the top of the stack.

Instruction `CALL` jumps to a new instruction sequence and creates a new region. It will be used in the compilation of a function application.

Instruction `PRIMOP` operates two basic values located in the stack and replaces them by the result of the operation.

Instruction `MATCH` performs a vectored jump depending on the constructor of the matched closure. The vector of sequences pointed to by the $\mathbf{p}_j$ corresponds to the compilation of a set of **case** alternatives. The `MATCH!` instruction additionally destroys the matched cell.

The `BUILDENV` instruction receives a list of keys K_i and creates a portion of environment on top of the stack: If a key K is a natural number j, the item $S!j$ is copied and pushed onto the stack; if it is a basic constant c, it is directly pushed onto the stack; if it is the identifier *self*, then the current region number k is pushed onto the stack.

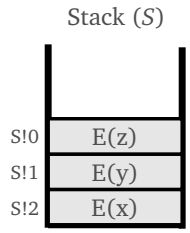
Instruction `BUILDCLS` allocates fresh memory and constructs a new cell in the style of the [*Cons*] rule in the big-step semantics. Similarly to `BUILDENV`, it receives a list of keys and uses the same conventions. It also receives the constructor C_r^m of the cell being created.

Finally, instruction `SLIDE` removes some parts of the stack. It will be used to remove environments when they are no longer needed.

5. Translating *Core-Safe* into SVM code

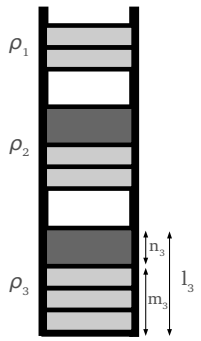
A major difference of the big-step operational semantics with respect to the SVM is the lack of a value environment E in the latter. The values of E are assumed to live in the stack S. In order to perform the translation from *Core-Safe* into SVM instructions, we need to set up a correspondence between program variables (which are no longer present during the execution of a SVM program) and the positions of the stack which contain the values of these variables.

The main idea of the translation is to keep a compile-time environment ρ mapping program variables to stack positions. As the stack grows dynamically, a first idea is to assign numbers to the variables from the bottom of the environment to the top. In this way, if the current environment occupies the top m positions of the stack and $\rho(x) = 1$, then $S!(m-1)$ will contain the runtime value corresponding to x.



A second idea is to reuse the current environment when pushing a continuation onto the stack. A continuation should contain the values of the variables that are in scope at the moment in which the continuation is pushed onto the stack. A naive implementation would create, from some of the elements of the stack, a value environment corresponding to the current context, and include that environment in the continuation. This leads to redundancies, as some of the elements in the current environment are located in the stack by themselves (i.e. outside continuations), while also being part of a continuation. Our aim is to *share* the current environment, instead of duplicating it, so that only two elements are kept in the continuation: the k_0 explained in Section 4, and a code pointer containing the expression which will be executed after popping the continuation.

In order to carry out this sharing between runtime environments, we split the whole compile-time environment ρ into a list of smaller environments $[\rho_1, \ldots, \rho_n]$, each one topped with a continuation, except the topmost one (ρ_1). Each individual block ρ_i consists of a triple (Δ_i, l_i, n_i) with the actual environment Δ_i mapping variables to numbers in the range $(1 \ldots m_i)$, its length $l_i = m_i + n_i$, and an indicator n_i whose value is 2 for all the blocks except for the first one, whose value is $n_1 = 0$. This value stands for



the length of the continuation $(k_0, \mathbf{p})$: we assume that a continuation needs two words in the stack and that the remaining items need one word. The positions given by the smaller environments Δ_i are relative to the bottommost position of their block in the stack.

Given this definition of a compile-time environment, we can compute the offset w.r.t. the top of the stack of a given variable x, defined in the k-th block. We use the notation $\rho(x)$ to denote this offset.

$$\rho(x) \stackrel{\text{def}}{=} \sum_{i=1}^{k} l_i - \Delta_k(x)$$

We assume by convention that $\rho(\mathit{self}) = \mathit{self}$. As soon as we introduce new variables in scope, we need to update the environment ρ accordingly. Only the topmost environment can be extended with new bindings. We define the following operations and functions on compile-time environments:

1. **Operator** $+$. It adds new bindings to the compile-time environment:

$$((\Delta, m, 0) : \rho) + [\overline{x_i \mapsto j_i}^n] \stackrel{\text{def}}{=} (\Delta \uplus [\overline{x_i \mapsto m + j_i}^n], m + n, 0) : \rho$$

2. **Operator** $^{++}$. It finishes the topmost block, and starts a new one above it:

$$((\Delta, m, 0) : \rho)^{++} \stackrel{\text{def}}{=} ([\,], 0, 0) : (\Delta, m + 2, 2) : \rho$$

3. **Function** *topDepth*. It returns the length of the topmost block. Its result is undefined when applied to empty environments.

$$\mathit{topDepth}((\Delta, m, 0) : \rho) \stackrel{\text{def}}{=} m$$

Given these conventions, in Figure 12 we show the translation functions *trE* and *trF* for translating expressions and function definitions, respectively. The *trE* function receives a *Core-Safe* expression, a compile-time environment ρ, and a function map F, which maps function names to code pointers. The latter specifies the SVM code to be executed when a call to a given function is done. The translation function returns list of SVM instructions and a code store. The expression *NormalForm* ρ is a compilation macro defined as follows: *NormalForm* ρ = `SLIDE` 1 (*topDepth*(ρ)) : `DECREGION` : `POPCONT`. This macro is used when the expression being translated gives rise to a normal form (i.e. no additional evaluations are needed). This is the case of literal,

$$
\begin{array}{lll}
trE\ c\ \rho\ F & = (\texttt{BUILDENV}\ [c] : \mathit{NormalForm}\ \rho\ ,\ [\,]) & \\
trE\ x\ \rho\ F & = (\texttt{BUILDENV}\ [\rho(x)] : \mathit{NormalForm}\ \rho\ ,\ [\,]) & \\
trE\ (x\ @\ r)\ \rho\ F & = (\texttt{BUILDENV}\ [\rho(x), \rho(r)] : \texttt{COPY} : \mathit{NormalForm}\ \rho\ ,\ [\,]) & \\
trE\ (a_1 \oplus a_2)\ \rho\ F & = (\texttt{BUILDENV}\ [\rho(a_1), \rho(a_2)] : \texttt{PRIMOP}\ \oplus : \mathit{NormalForm}\ \rho\ ,\ [\,]) & \\
trE\ (g\ \overline{a_i}^n\ @\ \overline{r_j}^m)\ \rho\ F & = (\texttt{BUILDENV}\ [\overline{\rho(a_i)}^n, \overline{\rho(r_j)}^m] : \texttt{SLIDE}\ (n+m)\ td : \texttt{CALL}\ \mathbf{p}\ ,\ [\,]) & \\
\quad \textbf{where} & \mathbf{p} = F(g) & \\
& td = \mathit{topDepth}(\rho) & \\
trE\ (C_l^m\ \overline{a_i}^n\ @\ r)\ \rho\ F & = (\texttt{BUILDCLS}\ C_l^m\ [\overline{\rho(a_i)}^n]\ (\rho(r)) : \mathit{NormalForm}\ \rho\ ,\ [\,]) & \\
trE\ (\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2)\ \rho\ F & = (\texttt{PUSHCONT}\ \mathbf{p} : is_1\ ,\ cs_1 \uplus cs_2 \uplus [\mathbf{p} \mapsto is_2]) & \\
\quad \textbf{where} & (is_1, cs_1) = trE\ e_1\ \rho^{++}\ F & \\
& (is_2, cs_2) = trE\ e_2\ (\rho + [x_1 \mapsto 1])\ F & \\
trE\ (\textbf{case}\ x\ \textbf{of}\ \overline{alt_i}^n)\ \rho\ F & = (\texttt{MATCH}\ (\rho(x))\ \overline{\mathbf{p}_i}^n\ ,\ (\biguplus_{i=1}^n cs_i) \uplus [\overline{\mathbf{p}_i \mapsto is_i}^n]) & \\
\quad \textbf{where} & \forall i \in \{1..n\}.(is_i, cs_i) = trA\ alt_i\ \rho\ F & \\
trE\ (\textbf{case!}\ x\ \textbf{of}\ \overline{alt_i}^n)\ \rho\ F & = (\texttt{MATCH!}\ (\rho(x))\ \overline{\mathbf{p}_i}^n\ ,\ (\biguplus_{i=1}^n cs_i) \uplus [\overline{\mathbf{p}_i \mapsto is_i}^n]) & \\
\quad \textbf{where} & \forall i \in \{1..n\}.(is_i, cs_i) = trA\ alt_i\ \rho\ F & \\
trA\ (C\ \overline{x_i}^n \to e_i)\ \rho\ F & = trE\ e_i\ (\rho + [\overline{x_i \mapsto n - i + 1}^n])\ F & \\
trF\ (f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f)\ F & = (cs \uplus [\mathbf{p} \mapsto is], F') & \\
\quad \textbf{where} & F' = F \uplus [f \mapsto \mathbf{p}] & \\
& (is, cs) = trE\ e_f\ [([\overline{r_j \mapsto m - j + 1}^m, \overline{x_i \mapsto n - i + m + 1}^n], n + m, 0)]\ F' &
\end{array}
$$

Figure 12: Translation from *Core-Safe* expressions to SVM instructions

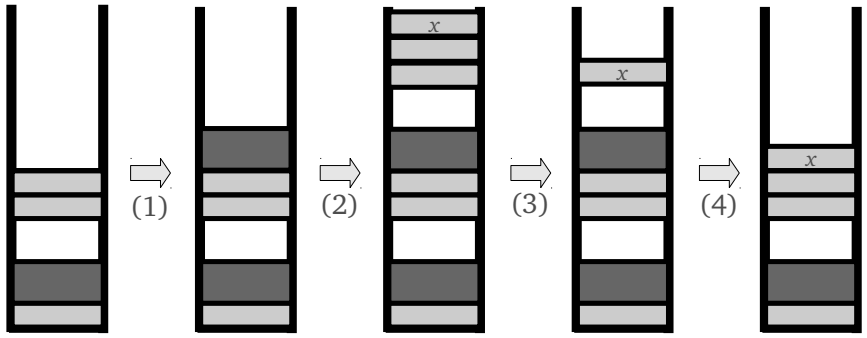


Figure 13: Evaluation of a **let** expression in the SVM. First, a continuation is inserted (1). Then, the evaluation of the auxiliary expression takes place until a normal form is reached and it is placed at the top of the stack (2). The previous environment is discarded (3) and the topmost continuation is removed, and the evaluation of the main expression begins.

variables, copy and constructor expressions, and also the case of primitive operators.

When evaluating a **let** expression (see Figure 13), a continuation is inserted before evaluating the auxiliary expression e_1. During translation, every binding added to the compile-time environment ρ will be added to a separate block, which is now the topmost one, until the translation of e_1 finishes. This corresponds to the removal of the topmost block and the continuation at runtime, so that the evaluation of the main expression e_2 begins.

An interesting case is that of function application. Firstly, the actual parameters are inserted into the stack. Since the execution of the function being called occurs in a different context, the previous environment can be discarded before the evaluation of the body of the function. The removal of the previous environment allows us to obtain constant stack space for tail-recursive functions, as the following example shows.

Example 16. Let us consider the following tail-recursive function definition for adding the elements of a list:

$$\begin{array}{lcl} \mathit{sumAc}\ \mathit{xs}\ \mathit{ac} & = & \textbf{case}\ \mathit{xs}\ \textbf{of} \\ & & \qquad [\,] \to \mathit{ac} \\ & & \qquad (x : \mathit{xx}) \to \textbf{let}\ x_1 = x + \mathit{ac}\ \textbf{in}\ \mathit{sumAc}\ \mathit{xx}\ x_1 \end{array}$$

The translation function *trE* applied to the body of *sumAc* and the function map $[sumAc \mapsto \mathbf{p}_{sumAc}]$ returns the following SVM code:

```
1  p_sumAc :  MATCH 0 [p_1, p_2]
2       p_1 :  BUILDENV [1]
3              SLIDE 1 2
4              DECREGION
5              POPCONT
6       p_2 :  PUSHCONT p_3
7              BUILDENV [2, 5]
8              PRIMOP +
9              SLIDE 1 0
10             DECREGION
11             POPCONT
12      p_3 :  BUILDENV [2, 0]
13             SLIDE 2 5
14             CALL p_sumAc
```

Let us assume we execute *sumAc* with the list $[5, 7]$ and the initial accumulator 0 given as input. In other words, we execute the SVM code starting from the $\mathbf{p}_{sumAc}$ label, with an initial heap $h = [p_1 \mapsto (1, (:)\, 5\, p_2), p_2 \mapsto (1, (:)\, 7\, p_3), p_3 \mapsto (1, [\,])]$, and assuming p_1 and 0 at the top of the evaluation stack. Figure 14 shows how the stack evolves during the execution of *sumAc*. The arrows between stacks contain the number of the SVM instruction being executed in each step (as labelled in the code above). In tail-recursive functions there are no continuations left in the stack when doing a recursive call. This means we can discard the whole environment in the current function call before proceeding to the next recursive call. As a result, *sumAc* runs in constant stack space. □

The *trF* function of Figure 12 deals with the translation of function definitions. It returns a code store and an updated function map with the code pointer of the definition being translated. This new binding is also passed to *trE*, in order to handle recursive calls. One may expect that the function map F given as input to *trF* associates the functions being called from e_f with the code pointers in a code store resulting from *trF* applied to these functions. This fact, which is crucial for proving the soundness of the translation, is given by the following definition.

Definition 17. A pair (F, cs) is said to be *generated from* Σ iff for every binding $[f \mapsto \mathbf{p}] \in F$, there exist n, m, $\overline{x_i}^n$, $\overline{r_j}^m$, e_f, $cs' \subseteq cs$, and $F' \subseteq F$ such that:

1. $(f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f) \in \Sigma$.
2. $trF\ (f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f)\ F' = (cs', F' \uplus [f \mapsto \mathbf{p}])$ for some $F' \subseteq F$.

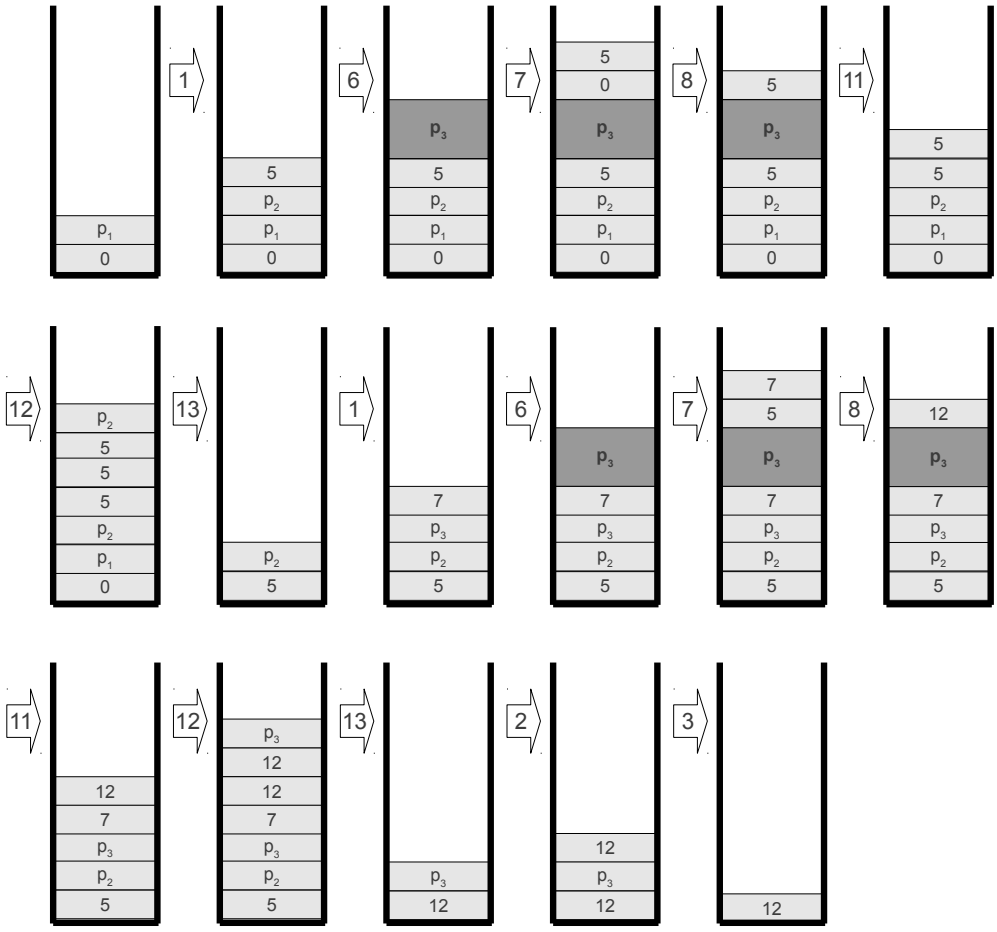


Figure 14: Contents of the stack during the execution of *sumAc*.

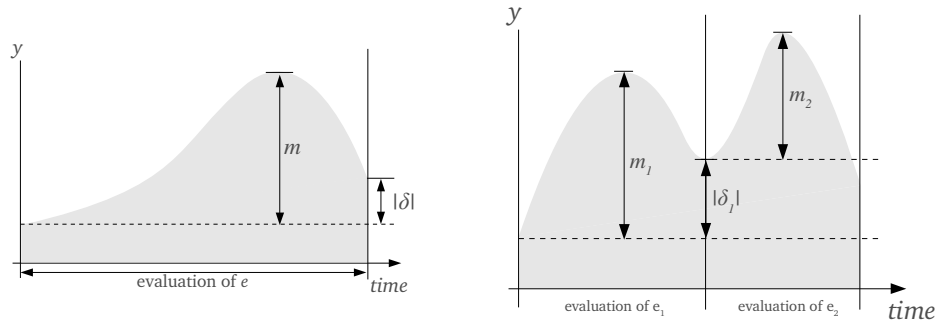


Figure 15: Intuitive meaning of δ and m components in the resource vector. The y coordinate represents the number of cells in the heap.

6. Resource-aware semantics

Once the resource consumption of each instruction of the SVM is known, we enrich the big-step semantics given in Section 3 with a resource vector (δ, m, s), which can be conceived as a side effect of evaluating e. The first component is a partial function $\delta : \mathbb{N} \to \mathbb{Z}$ giving, for each region k, the signed difference between the number of cells after and before evaluating the expression. A positive difference means that more cells have been created than destroyed. A negative one describes the opposite situation. The component m is a natural number describing the *minimum* number of fresh cells in the heap needed to successfully evaluate e. This number corresponds to the maximum amount of cells existing simultaneously in memory during the evaluation of this expression. The component s is a natural number whose meaning is analogous to that of the m component. The s component describes the minimum size of stack (in words) needed for the evaluation of the expression.

Figure 15a 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 point in time, after 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

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

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

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

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

$$\frac{\begin{array}{c}(g\ \overline{y_i}^n\ @\ \overline{r'_j}^l = e_g) \in \Sigma \\ [\overline{y_i \mapsto E(a_i)}^n, \overline{r'_j \mapsto E(r_j)}^l, self \mapsto k+1] \vdash (h,k+1), n+l, e \Downarrow (h',k+1), v, (\delta, m, s)\end{array}}{E \vdash (h,k), td, g\ \overline{a_i}^n\ @\ \overline{r_j}^l \Downarrow (h'\ |_k, k), v, (\delta\ |_k, m, \max\{n+l, s+n+l-td\})}\ [App]$$

$$\frac{E(r) = j \qquad j \leq k \qquad fresh_h(p)}{E \vdash (h,k), td, C\ \overline{a_i}^n\ @\ r \Downarrow (h \uplus [p \mapsto (j, C\ \overline{E(a_i)}^n)], k), v, ([j \mapsto 1]_k, 1, 1)}\ [Cons]$$

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

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

$$\frac{\begin{array}{c}C = C_r \qquad E(x) = p \qquad E \uplus [\overline{x_{ri} \mapsto v_i}^{n_r}] \vdash (h,k), td + n_r, e_r \Downarrow (h',k), v, (\delta, m, s) \\ \delta' = \delta + [j \mapsto -1]_k \qquad m' = \max\{0, m-1\}\end{array}}{E \vdash (h \uplus [p \mapsto (j, C\ \overline{v_i}^{n_r})], k), td, \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^n \Downarrow (h',k), v, (\delta', m', s + n_r)}\ [Case!]$$

Figure 16: Resource-aware operational semantics of *Core-Safe* expressions

represented in Figure 15a is the global balance $|\delta|$ of heap cells between the final and initial heaps, which will be 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 15a).

If we represented the stack consumption in the style of Figure 15a, the s component of the resource vector would take the role of the m component in the heap consumption. There is no component for denoting the difference in stack words between the final and initial heaps, because the final stack contains the same elements of the initial stack with an additional element at the top, which is the result of evaluating the expression, so this difference is always 1.

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 $[\ ]_k$ stands for the function $[i \mapsto 0 \mid i \in \{0..k\}]$, whereas $[i \mapsto n]_k$ 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:

$$|\delta| \stackrel{\text{def}}{=} \sum_{i \in \text{dom}\ \delta} \delta(i)$$

The notation $\delta_1 + \delta_2$ represents the componentwise addition of δ_1 and δ_2, provided these have the same domain:

$$\delta_1 + \delta_2 \stackrel{\text{def}}{=} [\delta_1(i) + \delta_2(i) \mid i \in \text{dom}\ \delta_1 \cap \text{dom}\ \delta_2]$$

The enriched semantic rules are shown in Figure 16. In addition to the resource vector (δ, m, s), we need a new component td in order to simulate the *topDepth* function of compile time environments. This component represents the number of stack words inserted after the topmost continuation, and it influences the s, since an amount td of words are removed from the stack before function calls.

The evaluation of an atom (rules $[Lit]$ and $[Var]$) does not require memory consumption, and it requires a stack word space to push the result onto. 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 defines this notion of size:

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

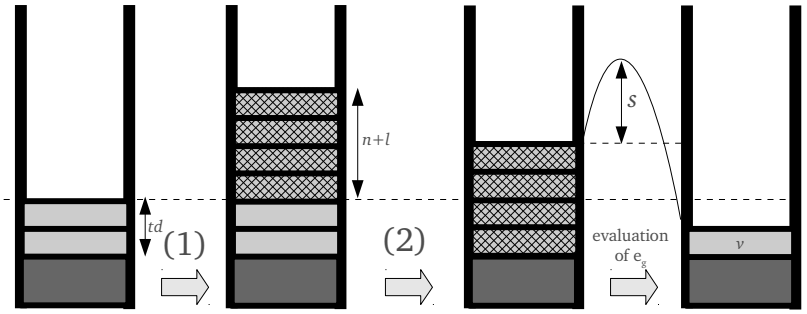


Figure 17: Stack consumption while evaluating a function application: we have to take the maximum between the number of arguments pushed onto the stack (1) and the maximum stack level reached during evaluation of the function's body, assuming that the arguments are already in the stack and the previous environment has been discarded (2).

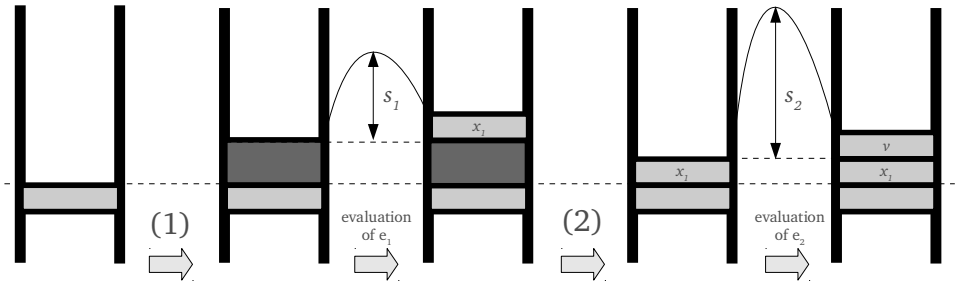


Figure 18: Stack consumption while evaluating a **let** expression: we have to take the maximum between the execution of e_1 assuming that a continuation has been pushed before onto the stack (1), and the execution of e_2 assuming that the value of x_1 has been pushed after discarding the continuation (2).

In rule $[App]$, 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 takes into account that the first $n+l$ words are needed to store the actual arguments, then the current environment of length td is discarded, and then the function body is evaluated (Figure 17).

In rule $[Let]$, a continuation (2 words) is stacked before evaluating e_1, and this leaves a value in the stack before evaluating e_2. That is why we obtain $\max\{2+s_1, 1+s_2\}$ as stack consumption (see Figure 18). With respect to the heap consumption, we take the maximum between the memory needs of e_1 and those of e_2, but taking the balance $|\delta_1|$ into account.

Example 18. In Figure 19 we show the resource vector corresponding to the execution of every example in this paper. The concrete input DSs are shown above the table. We assume that these DSs have been created before evaluating each function call, so their building costs are not reflected in the table.

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 *appendD* destroys four cells of the input list and builds three in the output region. Its execution needs no additional heap space. The heap cost of *insert* and *inssort* is proportional to the length of the input list, whereas in *insertD* and *inssortD* this cost is constant. In the case of *insertD* we need an additional cell for storing the new element. 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. Finally, the calls to *sumAc* produce no heap costs and a constant stack cost. □

7. Correctness of the translation into SVM

Now, we show that the pair translation-abstract machine is sound and complete with respect to the semantics defined in the last section. First, we note that both the semantics and the SVM machine rules are syntax driven, and that their computations are deterministic (up to fresh names generation).

Lemma 19. *Given a value environment E, an initial heap h_0 with k regions, a natural number td, and a Core-Safe expression e, if $E \vdash (h_0, k), td, e \Downarrow (h, k), v, (\delta, m, s)$ and $E \vdash (h_0, k), td, e \Downarrow (h', k), v', (\delta', m', s)$, then $h = h'$ (up to pointer renaming), $v = v'$, and $(\delta, m, s) = (\delta', m', s')$*

$xs = [1, 2, 3]$ (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	$[E(r) \mapsto 3]_k$	3	23
appendC xs ys @ r	$[E(r) \mapsto 6]_k$	6	24
appendD xs ys @ r	$[E(r) \mapsto -1]_k$	0	23
insert 10 *xs* @ r	$[E(r) \mapsto 5]_k$	5	32
insertD 10 *xs* @ r	$[E(r) \mapsto 1]_k$	1	32
inssort zs @ r	$[E(r) \mapsto 21]_k$	21	41
inssortD zs @ r	$[E(r) \mapsto 0]_k$	0	41
mkTree zs @ r	$[E(r) \mapsto 11]_k$	11	50
inorder t @ r	$[E(r) \mapsto 4]_k$	5	29
treesort zs @ r	$[E(r) \mapsto 6]_k$	18	62
sumAc xs 0	$[\,]_k$	0	6
sumAc zs 0	$[\,]_k$	0	6

Figure 19: Memory consumption results. The table shows, for each expression, the resource vector (δ, m, s) resulting from its evaluation.

Proof. By induction on the $\Downarrow$- derivation. All cases are straightforward. □

Lemma 20. *Given an initial SVM configuration c_{init} such that $c_{init} \to c_1$ and $c_{init} \to c'_1$. Then $c_1 = c'_1$ (modulo pointer renaming).*

Proof. This can be proved by case distinction on the rule being applied. All cases are straightforward. □

The main difference between the big-step operational semantics of Section 3 and the SVM machine is the way in which value environments are represented. In the big-step semantics we have a mapping E from variables to values, whereas in the SVM we have a stack. The correspondence between variables and positions of the stack is given by the ρ environment used in the translation (see Section 5).

Obviously, we cannot ensure the equivalence of big-step semantics and SVM if their starting points (i.e. mapping E in big-step semantics, stack S in the SVM) denote different value environments. This is the motivation for the following:

Definition 21. We say that the environment E and the pair (ρ, S) are equivalent, denoted $E \equiv (\rho, S)$, if $\mathrm{dom}\, E = \mathrm{dom}\, \rho$, and $\forall x \in \mathrm{dom}\, \rho \backslash \{self\}.E(x) = S!(\rho(x))$.

A stack S' is said to be a *suffix* of S if there exists a number $n \geq 0$ of stack elements b_i such that $S = \overline{b_i}^n : S'$. Given a SVM configuration $c_0 = (is, h, k_0, k, S, cs)$ and S' a suffix of S, we denote by $c_0 \rightarrow^*_{S'} c_n$ a derivation in which all the stacks in intermediate configurations have S' as a suffix. Should the top instruction of a configuration create a stack smaller than S', then the machine would stop at that configuration.

Now, we follow with some definitions regarding the maximum memory consumption produced by a SVM program.

Definition 22. The function *sizeST*, which returns the size (in words) of a stack, is defined as follows:

$$\begin{array}{ll} sizeST([\,]) & = 0 \\ sizeST(v : S) & = 1 + sizeST(S) \text{ if } v \text{ is not a continuation.} \\ sizeST((k_0, \mathbf{p}) : S) & = 2 + sizeST(S) \end{array}$$

Analogously, the size (in cells) of a heap is given by the function *sizeH*:

$$sizeH(h) = |\mathrm{dom}\, h|$$

We shall extend the notation of these functions to configurations, so $sizeST(c)$ (resp. $sizeH(c)$) will be used to denote the size of the stack (resp. heap) of the given configuration c.

Given $c_0 = (is, h, k_0, k, S, cs)$ and a derivation $c_0 \rightarrow_{S'} \cdots \rightarrow_{S'} c_n$, the *maximum number of fresh cells* of the derivation, denoted $maxFreshCells\ (c_0 \rightarrow^*_{S'} c_n)$, is the highest difference in cells between the heaps of the configurations $c_0, \ldots c_n$ and the initial heap h. Likewise, we define the *maximum number of fresh words* created in the stack S, denoted $maxFreshWords(c_0 \rightarrow^*_{S'} c_n)$. Finally, by $diff\,(k, h, h')$ we denote a function giving for each region in $\{0, \ldots, k\}$ the signed difference in cells between h' and h.

Definition 23. Given a derivation $c_0 \rightarrow_{S'} \ldots c_i \ldots \rightarrow_{S'} c_n$, we define:

$$\begin{array}{rcl} maxFreshCells(c_0 \rightarrow^*_{S'} c_n) & = & \max\{sizeH(c_i) - sizeH(c_0) \mid 0 \leq i \leq n\} \\ maxFreshWords(c_0 \rightarrow^*_{S'} c_n) & = & \max\{sizeST(c_i) - sizeST(c_0) \mid 0 \leq i \leq n\} \end{array}$$

From these definitions the following properties can be easily obtained:

$$\begin{aligned} \mathit{maxFreshCells}(c_0 \to^*_{S'} c_i \to^*_{S'} c_n) = \max\{ \quad & \mathit{maxFreshCells}(c_0 \to^*_{S'} c_i), \\ & \mathit{maxFreshCells}(c_i \to^*_{S'} c_n) \\ & \quad + \mathit{sizeH}(c_i) - \mathit{sizeH}(c_0) \} \end{aligned}$$

$$\begin{aligned} \mathit{maxFreshWords}(c_0 \to^*_{S'} c_i \to^*_{S'} c_n) = \max\{ \quad & \mathit{maxFreshWords}(c_0 \to^*_{S'} c_i), \\ & \mathit{maxFreshWords}(c_i \to^*_{S'} c_n) \\ & \quad + \mathit{sizeST}(c_i) - \mathit{sizeST}(c_0) \} \end{aligned}$$

Definition 24. Given two heaps h and h' and a number k of regions, we denote by $\mathit{diff}(k, h, h')$ a function δ such that $\mathrm{dom}\ \delta = \{0..k\}$ and for all $i \in \{0..k\}$:

$$\delta(i) = |\{p \in \mathrm{dom}\ h' \mid \mathit{region}(h'(p)) = i\}| - |\{p \in \mathrm{dom}\ h \mid \mathit{region}(h(p)) = i\}|$$

The definitions of *diff* and *sizeH* are related by the following property, which is easy to establish from the corresponding definitions. Let h and h' be two heaps with k regions. Then:

$$\mathit{sizeH}(h') - \mathit{sizeH}(h) = \sum_{i=0}^{k} \mathit{diff}(i, h, h')$$

The next Lemma shows some properties of the *copy* function. In particular, that the size of the copy and the DS being copied are the same.

Lemma 25. *If* $(h', p') = \mathit{copy}(h, p, j)$ *then:*

1. $h \subseteq h'$
2. $\mathit{size}(h', p') = \mathit{size}(h, p)$
3. $\mathit{sizeH}(h') - \mathit{sizeH}(h) = \mathit{size}(h', p')$
4. $\forall k \geq j,\ \mathit{diff}(k, h, h') = [j \mapsto \mathit{size}(h', p')]_k$

Proof. Straightforward induction on the size of the DS pointed by p. □

The following theorem establishes the correctness of the translation, showing that the computed value and the resource consumption of a given expression e in the big-step semantics are the same as those obtained by executing in the SVM the translation of e.

We denote by $\mathit{drop}\ n\ S$ the stack resulting from removing the n topmost elements of S. That is, $\mathit{drop}\ n\ (\overline{b_i}^n : S) = S$ and undefined in case the number of elements in the input stack is less than n.

Theorem 26. *For all $S, S', E, h, h', td, k_0, k, e, v, \delta, m, s, \rho, cs, cs'$, F, and Σ of their respective types, if*

$$\begin{array}{lll} E \text{ admissible w.r.t. } k & (F, cs') \text{ generated from } \Sigma & \\ E \equiv (\rho, S) & (is, cs) = trE\ e\ \rho\ F & td = topDepth(\rho) \\ S' = drop\ td\ S & cs' \supseteq cs & k_0 \leq k \end{array}$$

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

1. $\underbrace{(is, h, k_0, k, S, cs')}_{c_{init}} \rightarrow^*_{S'} \underbrace{([\texttt{POPCONT}], h' \mid_{k_0}, k_0, k_0, v : S', cs')}_{c_{final}}$
2. $\delta = diff(k, h, h')$
3. $m = maxFreshCells(c_{init} \rightarrow^*_{S'} c_{final})$
4. $s = maxFreshWords(c_{init} \rightarrow^*_{S'} c_{final})$

Proof. The ($\Rightarrow$) direction is shown by induction on the $\Downarrow$-derivation. The ($\Leftarrow$) direction of the theorem can be proved by induction on the length of the $\rightarrow^*_{S'}$ derivation. In both proofs we distinguish cases depending on the expression being translated. Here we show only two representative cases. The complete proof can be found in the Appendix.

- **Case** [*Var*]: $e \equiv x$

  Let $(is, cs) = trE\ e\ \rho\ F$ be the SVM code generated, where $cs = [\,]$ and:

$$is = \texttt{BUILDENV}\ [\rho(x)] : \overbrace{\texttt{SLIDE}\ 1\ (topDepth(\rho)) : \underbrace{\texttt{DECREGION} : \underbrace{[\texttt{POPCONT}]}_{is_3}}_{is_2}}^{is_1}$$

  ($\Rightarrow$) Assume $E[x \mapsto v] \vdash (h, k), td, x \Downarrow (h, k), v, ([\,]_k, 0, 1)$ holds.
  For any cs' such that $cs' \supseteq cs$, the code generated leads to the following SVM derivation:

$$\begin{array}{ll} & \overbrace{(\texttt{BUILDENV}\ [\rho(x)] : is_1, \quad h, \quad k_0, \quad k, \quad S, \quad cs')}^{c_{init}} \\ \rightarrow_{S'} & \{\text{since } S!(\rho(x)) = E(x) = v\} \end{array}$$

$$
\begin{array}{ll}
 & (\texttt{SLIDE}\ 1\ (\mathit{topDepth}(\rho)) : is_2,\ \ h,\ \ k_0,\ \ k,\ \ v : S,\ \ cs') \\
\rightarrow_{S'} & \quad \{\text{since } td = \mathit{topDepth}(\rho)\} \\
 & (\texttt{DECREGION} : is_3,\ \ h,\ \ k_0,\ \ k,\ \ v : \mathit{drop}\ td\ S,\ \ cs') \\
\rightarrow_{S'} & \\
 & \underbrace{([\texttt{POPCONT}],\ \ h|_{k_0},\ \ k_0,\ \ k_0,\ \ v : \mathit{drop}\ td\ S,\ \ cs')}_{c_{\mathit{final}}}
\end{array}
$$

Consequently, property (1) holds. Property (2) holds because $\mathit{diff}(k, h, h) = [i \mapsto 0 \mid 0 \leq i \leq k] = \delta$. Property (3) and (4) hold because $\mathit{maxFreshCells}(c_{\mathit{init}} \rightarrow^*_{S'} c_{\mathit{final}}) = 0$ and $\mathit{maxFreshWords}$ $(c_{\mathit{init}} \rightarrow^*_{S'} c_{\mathit{final}}) = 1$.

(⇐) Given any cs', the only possible derivation is the one above. Trivially, by rule $[\mathit{Var}]$, $E[x \mapsto v] \vdash (h, k), td, x \Downarrow (h, k), v, ([\,]_k, 0, 1)$. Properties (2), (3) and (4) hold for the same reasons as in the opposite implication.

- **Case** $[\mathit{Let}]$: $e \equiv \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2$

  Assume $(is_1, cs_1) = \mathit{trE}\ e_1\ \rho^{+\!\!+}\ F$ and $(is_2, cs_2) = \mathit{trE}\ e_2\ \rho_1\ F$ where $\rho_1 = \rho + [x_1 \mapsto 1]$. On the other hand, we get $cs \supseteq cs_1$, $cs \supseteq cs_2$ and $is = \texttt{PUSHCONT}\ \mathbf{p} : is_1$, in which $[\mathbf{p} \mapsto is_2] \in cs \subseteq cs'$. Hence $cs'(\mathbf{p}) = is_2$.

  (⇒) Given $cs' \supseteq cs$, the trace corresponding to the execution of e starts as follows:

$$
\begin{array}{ll}
 & \overbrace{(\texttt{PUSHCONT}\ \mathbf{p} : is_1,\ \ h,\ \ k_0,\ \ k,\ \ S,\ \ cs')}^{c_{\mathit{init}}} \\
\rightarrow_{S'} & \\
 & (is_1,\ \ h,\ \ k,\ \ k,\ \ (k_0, \mathbf{p}) : S,\ \ cs') \equiv c_1
\end{array}
$$

  By rule $[\mathit{Let}]$ we get:

$$
\begin{array}{l}
E \vdash (h, k), 0, e_1 \Downarrow (h', k), v_1, (\delta_1, m_1, s_1) \\
E_1 \vdash (h', k), td + 1, e_2 \Downarrow (h'', k), v, (\delta_2, m_2, s_2)
\end{array}
$$

  where $E_1 = E \uplus [x_1 \mapsto v_1]$. In order to apply the induction hypothesis on the $\Downarrow$-derivation of e_1 we have to check that $E \equiv (\rho^{+\!\!+}, (k_0, \mathbf{p}) : S)$, which trivially holds from the hypothesis $E \equiv$

(ρ, S) and because dom ρ = dom $\rho^{+\!+}$. The rest of the assumptions hold trivially.

In this case $td = 0$, so by applying i.h. we obtain:

$$
\begin{array}{ll}
 & c_1 \\
\to^*_{(k_0,\mathbf{p}):S} & \\
 & ([\texttt{POPCONT}],\ h'|_k,\ k,\ k,\ v_1 : \mathit{drop}\ 0\ ((k_0,\mathbf{p}) : S),\ cs'[\mathbf{p} \mapsto is_2]) \equiv c_2
\end{array}
$$

which trivially implies $c_1 \to^*_{S'} c_2$.

Consequently:

$$
\begin{array}{ll}
 & \overbrace{(\texttt{PUSHCONT}\ \mathbf{p} : is_1,\ \ h,\ \ k_0,\ \ k,\ \ S,\ \ cs')}^{c_{init}} \\
\to_{S'} & \\
 & (is_1,\ \ h,\ \ k,\ \ k,\ \ (k_0,\mathbf{p}) : S,\ \ cs') \equiv c_1 \\
\to^*_{S'} & \quad \{\ \text{since } cs'(\mathbf{p}) = is_2\ \} \\
 & ([\texttt{POPCONT}],\ \ h'|_k,\ k,\ k,\ v_1 : \mathit{drop}\ 0\ ((k_0,\mathbf{p}) : S),\ cs'[\mathbf{p} \mapsto is_2]) \equiv c_2 \\
\to_{S'} & \\
 & (is_2,\ \ h'|_k,\ \ k_0,\ \ k,\ \ v_1 : S,\ \ cs') \equiv c_3 \\
\to^*_{S'} & \quad \{\ \text{by i.h. (see below)}\ \} \\
 & ([\texttt{POPCONT}],\ \ h''|_{k_0},\ \ k_0,\ \ k_0,\ \ v : \mathit{drop}\ (td+1)\ (v_1 : S),\ \ cs') \\
\equiv & \\
 & \underbrace{([\texttt{POPCONT}],\ \ h''|_{k_0},\ \ k_0,\ \ k_0,\ \ v : S',\ \ cs')}_{c_{final}}
\end{array}
$$

In order to apply the induction hypothesis on the $\Downarrow$-derivation of e_2 in the last step we have to check that $E_1 \equiv (\rho_1, v_1 : S)$. Let $x \in$ dom E_1: if $x = x_1$ then $(v_1 : S)!(\rho_1(x)) = (v_1 : S)!0 = v_1 = E_1(x)$. If $x \neq x_1$ then $(v_1 : S)!(\rho_1(x)) = (v_1 : S)!(\rho(x) + 1) = S!(\rho(x)) = E(x) = E_1(x)$. Additionally, $\mathit{drop}\ (td+1)\ (v_1 : S) = \mathit{drop}\ td\ S = S'$. The rest of the assumptions hold trivially.

Hence (1) holds. With respect to (2), let $i \in \{1..k\}$

$$
\begin{array}{l}
(\mathit{diff}(k,h,h''))(i) \\
\quad = |\{p \in h'' \mid \mathit{region}(h''(p)) = i\}| - |\{p \in h \mid \mathit{region}(h(p)) = i\}| \\
\quad = |\{p \in h'' \mid \mathit{region}(h''(p)) = i\}| - |\{p \in h' \mid \mathit{region}(h'(p)) = i\}| \\
\qquad + |\{p \in h' \mid \mathit{region}(h'(p)) = i\}| - |\{p \in h \mid \mathit{region}(h(p)) = i\}| \\
\quad = (\mathit{diff}(k,h,h'))(i) + (\mathit{diff}(k,h',h''))(i) \\
\quad = \delta_1(i) + \delta_2(i)
\end{array}
$$

Therefore $\mathit{diff}(k, h, h'') = \delta_1 + \delta_2$. Properties (3) and (4) are proven as follows:

$$
\begin{aligned}
&\mathit{maxFreshCells}(c_{\mathit{init}} \rightarrow^*_{S'} c_2) \quad = m_1 \\
&\mathit{maxFreshCells}(c_2 \rightarrow^*_{S'} c_{\mathit{final}}) \quad = m_2 \\
&\mathit{maxFreshCells}(c_{\mathit{init}} \rightarrow^*_{S'} c_{\mathit{final}}) \\
&\quad = \max\{m_1, m_2 + \mathit{sizeH}(c_2) - \mathit{sizeH}(c_{\mathit{init}})\} \\
&\quad = \max\{m_1, m_2 + \textstyle\sum_{i=0}^{k}(\mathit{diff}(k, h, h'))(i)\} \\
&\quad = \max\{m_1, m_2 + |\delta_1|\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{maxFreshWords}(c_{\mathit{init}} \rightarrow^*_{S'} c_1) = 2 \\
&\mathit{maxFreshWords}(c_{\mathit{init}} \rightarrow^*_{S'} c_2) \\
&\quad = \max\ \{2, 2 + s_1\} = 2 + s_1 \\
&\mathit{maxFreshWords}(c_{\mathit{init}} \rightarrow^*_{S'} c_2 \rightarrow^*_{S'} c_3) \\
&\quad = \max\ \{2 + s_1, 1\} = 2 + s_1 \\
&\mathit{maxFreshWords}(c_{\mathit{init}} \rightarrow^*_{S'} c_3 \rightarrow^*_{S'} c_{\mathit{final}}) \\
&\quad = \max\ \{2 + s_1, 1 + s_2\}
\end{aligned}
$$

($\Leftarrow$) Given $cs' \supseteq cs$, the trace corresponding to the execution of e must be as follows:

$$
\begin{aligned}
&\overbrace{(\texttt{PUSHCONT}\ \mathbf{p} : is_1,\quad h,\quad k_0,\quad k,\quad S,\quad cs')}^{c_{\mathit{init}}} \\
\rightarrow_{S'}\ &(is_1,\quad h,\quad k,\quad k,\quad (k_0, \mathbf{p}) : S,\quad cs') \equiv c_1 \\
\rightarrow^*_{S'}\ &\{\text{by Lemma 27 (see Appendix)}\} \\
&([\texttt{POPCONT}],\quad h',\ k,\ k,\ v_1 : (k_0, \mathbf{p}) : S),\ cs'[\mathbf{p} \mapsto is_2]) \equiv c_2 \\
\rightarrow_{S'}\ &(is_2,\quad h',\quad k_0,\quad k,\quad v_1 : S,\quad cs') \equiv c_3 \\
\rightarrow^*_{S'}\ &\underbrace{([\texttt{POPCONT}],\quad h'',\quad k_0,\quad k_0,\quad v : S',\quad cs')}_{c_{\mathit{final}}}
\end{aligned}
$$

Notice that c_1 contains a continuation $(k_0, \mathbf{p})$ on top of the stack. The only machine instruction which removes such continuation from the stack is `POPCONT`. So, in order to reach c_{final} an intermediate configuration $c_2 \equiv ([\texttt{POPCONT}],\quad h',\ k',\ k',\ v_1 : (k_0, \mathbf{p}) :$

$S)$, $cs'[\mathbf{p} \mapsto is_2]$) must be reached so that the execution can proceed. By Lemma 27 in Appendix A, $k' = k$.

In fact, it holds that $c_1 \rightarrow^*_{(k_0,\mathbf{p}):S} c_2$, so we can apply i.h. to obtain $E \vdash (h,k), 0, e_1 \Downarrow (h',k), v_1, (\delta_1, m_1, s_1)$, as $(k_0, \mathbf{p}) : S = \mathit{drop}\ 0\ ((k_0, \mathbf{p}) : S)$.

Notice also that $E \uplus [x_1 \mapsto v_1] \equiv (\rho + [x_1 \mapsto 1], v_1 : S)$ and that $\mathit{drop}\ (td+1)\ (v_1 : S) = \mathit{drop}\ td\ S = S'$, so we can also apply i.h. to $c_3 \rightarrow^*_{S'} c_{final}$ to obtain $E \uplus [x_1 \mapsto v_1] \vdash (h',k), td+1, e_2 \Downarrow (h'',k), v, (\delta_2, m_2, s_2)$. The reasoning about resources consumption is the same as the opposite implication.

□

8. Related Work and Conclusions

The motivation for this work has been to show part of the implementation of *Safe*. One contribution is a systematic method for refining operational semantics and abstract machines in order to find the way from an abstract view of the language to an efficient implementation. Other contributions include a semantics enriched with memory costs, and the proof of correctness of these costs when translating *Safe* to imperative code. This resource-aware semantics is the basis for proving correct the memory consumption static analysis presented in [6], and for certifying these bounds, done in [8].

8.1. Regions and Deallocation of Cells

The use of regions in functional languages to avoid garbage collection is not new. Tofte and Talpin [14] introduced in MLKit (a variant of ML) the use of nested regions by means of a **letregion** construct. A lot of work has been done on this system [15, 16, 17]. Their main contribution is a *region inference* algorithm adding region annotations at the intermediate language level. A small difference with these approaches is that, in *Safe*, region allocation and deallocation are synchronized with function calls instead of being introduced by a special language construct. This simplifies the process of inferring regions, as explained in [5]. However, this comes at the cost of granularity for determining the region scopes: MLKit allows several lexically-scoped regions in the same function. A more relevant difference is that *Safe* has an additional mechanism allowing the programmer to selectively destroy data structures inside a region.

A difficulty with Tofte and Talpin's original system is the fact that regions have nested lifetimes. There exist a few programs (such as the *inssort* function shown in Example 11) that may result in memory leaks due to this restriction. In [18] this problem is alleviated by defining a variant of λ-calculus with type-safe primitives for creating, accessing and destroying regions, which are not restricted to have nested lifetimes. Programs are written in a C-like language called *Cyclone* having explicit memory management primitives, then they are translated into this variant of λ-calculus, and then type checked. So, the price of this flexibility is explicit region control. In our language *Safe*, regions also suffer from the nested lifetimes constraint, since both region allocation and deallocation are bound to function calls, which are necessarily nested. However, the destructive pattern matching facility compensates for this, since it is possible to dispose of a data structure without deallocating the whole region in which it resides. Allocation and destruction of distinct data structures are not necessarily nested, and the type system presented in [3, 4] protects the programmer against missuses of this feature. Again, the price of this flexibility is explicit deallocation of cells. Allocation is implicit in constructions and the target region of the allocation is inferred by the compiler. It is arguable whether it is better to explicitly manage regions or cells. In [19, 20] a more detailed comparison with all these works can be found.

The destructive pattern matching features of the language have been inspired by Hofmann and Jost's match construct [21], whose operational behaviour is similar to that of *Safe*'s **case!**. The main difference is that they lack a compile time analysis guaranteeing the safe use of this dangerous feature, since it is beyond the scope of their work.

8.2. Semantics and Abstract Machines

There have been other successful derivations of abstract machines starting from high level descriptions of the semantics. For instance, in [22] and [23] a number of such derivations are done. Well known abstract machines for the λ-calculus such as SECD, Krivine's, CLS and CAM are derived and proved correct. These papers propose general schemes for achieving this kind of derivations. The differences with the present work are the following:

- They concentrate on the pure λ-calculus and they consider neither sharing nor heaps. Algebraic types, **case** and **let** expressions are not considered either.

- In the second paper, the starting point is a denotational meaning of the source language, while here we start from an operational semantics.
- In order to refine their machines they use predefined correct transformations such as closure conversion, transformation into continuation passing style, defunctionalization and inlining.
- They ignore the compilation issues from the source language to machine instructions, and also resource consumption.

In [24] a broad survey of both abstract and virtual machines for the λ-calculus and for practical functional languages is done. The author presents in detail some well-known and other less known abstract machines. When the machines execute compiled code, also the translation schemes are provided. The aim of the book is to serve as a text for a graduate course and no attempt is done to provide proofs of correctness either of the machines or of the compilation schemes.

We have found inspiration in Sestoft's derivation of abstract machines for a lazy λ-calculus [25]. One of the authors has reported some previous experience in [26], but in that occasion the destination machine was known in advance. The present work represents a 'real' derivation in the sense that the destination machine has been invented from scratch.

Compared to other eager machines such as Landin's SECD machine [27], it is an added value of our abstract machine that the standard translation yields constant stack space for tail recursion, as we have shown in Example 16. For instance, in the G-machine the compiler needs to explicitly identify tail recursion and to do a special translation in this case, i.e. it is considered as an optimization of the code generation phase. The same happens in other compiled virtual machines such as π-RED [28]. Additionally, our SVM machine does not need an additional garbage collector and all memory allocation/deallocation actions have been implemented in constant time.

For the semantics enriched with a resource vector, we have found inspiration in [29]. Some other resource-enriched semantics have been proposed. See for example [30, 31] for a big-step semantics of a simple functional language with some information about the number of cells needed by an expression, while [32] extends the same idea to a higher-order language. In [33] Shkaravska et al. develop a type-based analysis for inferring information about polynomial input-output size dependencies. At runtime, the size of the data

structures can be obtained from the heap in a similar way to our *size* function. However, in our resource semantics we need to carry around additional information. In particular, the m and s components depend on the history of the execution, and not only on the final state.

Using some form of formal verification to ensure the correctness of compilers has been a hot topic for many years. An annotated bibliography covering up to 2003 can be found in [34]. Most of the papers reflected there propose techniques whose validity is established by formal proofs made and read by humans. Using machine-assisted proofs for compilers starts around the seventies, with an intensification at the end of the nineties.

Related to our work are [35] which certifies the translation of a Lisp subset to a stack language by using PVS, and [36] which uses Isabelle/HOL to formalise the translation from a small subset of Java (called μ-Java) to a stripped version of the Java Virtual Machine (17 bytecode instructions). Both specify the translation functions, and prove correctness theorems similar to ours. The latter work can be considered as a first attempt on Java, and it was considerably extended by Klein, Nipkow, Berghofer, and Strecker himself in [37, 38, 39].

A realistic C compiler for programming embedded systems has been built and verified in [40, 41, 42]. The source is a small C subset called *Cminor* to which C is informally translated, and the target is Power PC assembly language. The compiler runs through six intermediate languages for which the semantics are defined and the translation pass verified. The authors use the Coq proof-assistant and its extraction facilities to produce Caml code. This is perhaps the biggest project on machine-assisted compiler verification done up to now.

The current implementation of *Safe* generates either C or Java bytecode as a result. A strong point of the Java virtual machine is that it is available in many software and hardware platforms. However, it is somewhat restrictive with regard to memory management. As a consequence of this, the explicit destruction of a cell is handled by linking that cell to a free list. This cell is reused by the runtime system when a subsequent allocation takes place. Moreover, the statically typed nature of the JVM makes the process of translation awkward. It is convenient to target the bytecode generation to other virtual machines and compilation frameworks which allow more flexibility in memory management. A notable example of this is the LLVM (*Low Level Virtual Machine*) compilation framework [43], which provides a language-independent instruction set. Programs written in this language can

be further translated into machine code.

References

[1] G. C. Necula, Proof-Carrying Code, in: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97, ACM Press, 1997, pp. 106–119.

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

[3] 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 Conference on Principles and Practice of Declarative Programming, PPDP'08, ACM Press, 2008, pp. 152–162.

[4] M. Montenegro, R. Peña, C. Segura, An Inference Algorithm for Guaranteeing Safe Destruction, in: Selected papers of the 18th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR'08, LNCS 5438, Springer, 2009, pp. 135–151.

[5] M. Montenegro, R. Peña, C. Segura, A Simple Region Inference Algorithm for a First-order Functional Language, in: Selected papers of the 18th International Workshop on Functional and (Constraint) Logic Programming, WFLP'09, LNCS 5979, Springer, 2009, pp. 145–161.

[6] M. Montenegro, R. Peña, C. Segura, A Space Consumption Analysis By Abstract Interpretation, in: Selected papers of the 1st International Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA'09, LNCS 6324, Springer, 2009, pp. 34–50.

[7] J. de Dios, M. Montenegro, R. Peña, Certified Absence of Dangling Pointers in a Language with Explicit Deallocation, in: Proceedings of the 8th International Conference on Integrated Formal Methods, IFM'10, LNCS 6396, Springer, 2010, pp. 305–319.

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

[9] J. de Dios, R. Peña, Formal Certification of a Resource-Aware Language Implementation, in: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOL'09, LNCS 5674, Springer, 2009, pp. 196–211.

[10] J. de Dios, R. Peña, A Certified Implementation on top of the Java Virtual Machine, in: Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems, FMICS'09, LNCS 5825, Springer, 2009, pp. 1–16.

[11] M. Montenegro, R. Peña, C. Segura, A Resource-Aware Semantics and Abstract Machine for a Functional Language with Explicit Deallocation, in: Selected papers of the 17th Workshop on Functional and (Constraint) Logic Programming, WFLP'08, ENTCS 246, Elsevier, 2009, pp. 167–182.

[12] T. Nipkow, L. Paulson, M. Wenzel, Isabelle/HOL. A Proof Assistant for Higher-Order Logic, LNCS 2283, Springer, 2002.

[13] 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.

[14] M. Tofte, J.-P. Talpin, Region-based memory management, Information and Computation 132 (1997) 109–176.

[15] A. Aiken, M. Fähndrich, R. Levien, Better static memory management: Improving region-based analysis of higher-order languages, in: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation (PLDI'95), pp. 174–185.

[16] L. Birkedal, M. Tofte, M. Vejlstrup, From region inference to von Neumann machines via region representation inference, in: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 1996, pp. 171–183.

[17] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, P. Sestoft, Programming with regions for the MLKit, Technical Report, University of Copenhagen, 2006.

[18] M. Fluet, G. Morrisett, A. Ahmed, Linear regions are all you need, in: 15th European Symposium on Programming, ESOP 2006, Springer, 2006, pp. 7–21.

[19] R. Peña, C. Segura, M. Montenegro, A Sharing Analysis for SAFE, in: Trends in Functional Programming (Volume 7). Selected Papers of the 7th Symposium on Trends in Functional Programming, TFP'06, Intellect, 2007, pp. 109–128.

[20] M. Montenegro, Safety properties and memory bound analysis in a functional language without a garbage collector, Ph.D. thesis, Facultad de Informática, Universidad Complutense de Madrid, 2011.

[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, pp. 185–197.

[22] J. Hannan, D. Miller, From Operational Semantics to Abstract Machines, Mathematical Structures in Computer Science 2 (1992) 415–459.

[23] M. S. Ager, D. Biernacki, O. Danvy, J. Midtgaard, A Functional Correspondence between Evaluators and Abstract Machines, in: Proceedings of the 5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP'03, ACM Press, 2003, pp. 8–19.

[24] W. Kluge, Abstract Computing Machines: A Lambda Calculus Perspective, Springer Texts in Theoretical Computer Science, 2005.

[25] P. Sestoft, Deriving a Lazy Abstract Machine, Journal of Functional Programming 7 (3), Cambridge University Press (1997) 231–264.

[26] A. de la Encina, R. Peña, From Natural Semantics to C: A Formal Derivarion of two STG Machines, Journal of Functional Programming 19 (1), Cambridge University Press (2008) 47–94.

[27] P. Landin, The Mechanical Evaluation of Expressions, Computer Journal 6 (4) (1964) 308–320.

[28] D. Gaertner, W. Kluge, π-RED$^+$ – an Interactive Compiling Graph Reduction System for an Applied λ-Calculus, Journal of Functional Programming 6 (5) (1996) 723–757.

[29] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, A. Momigliano, A program logic for resources, Theoretical Computer Science 389 (2007) 411–445.

[30] M. Hofmann, S. Jost, Static prediction of heap space usage for first-order functional programs, in: Proceedings of the 30th ACM Symposium on Principles of Programming Languages, POPL'03, ACM Press, 2003, pp. 185–197.

[31] J. Hoffmann, M. Hofmann, Amortized Resource Analysis with Polynomial Potential. A Static Inference of Polynomial Bounds for Functional Programs, in: Proceedings of the 19th European Symposium on Programming, ESOP'10, LNCS 6012, Springer, 2010, pp. 287–306.

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

[33] O. Shkaravska, M. C. J. D. van Eekelen, R. van Kesteren, Polynomial size analysis of first-order shapely functions, Logical Methods in Computer Science 5 (2009).

[34] M. A. Dave, Compiler verification: a bibliography, SIGSOFT Software Engineering Notes 28 (2003) 2–2.

[35] A. Dold and V. Vialard, A Mechanically Verified Compiling Specification for a Lisp Compiler, in: Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'01, LNCS 2245, Springer, 2001, pp. 144–155.

[36] M. Strecker, Formal Verification of a Java Compiler in Isabelle, in: CADE-18, 18th International Conference on Automated Deduction, CADE'02, LNCS 2392, Springer, pp. 63–77.

[37] G. Klein, T. Nipkow, Verified Bytecode Verifiers, Theoretical Computer Science 298 (2003) 583–626.

[38] G. Klein, T. Nipkow, A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler, ACM Transactions on Programming Languages and Systems 28 (4) (2006) 619–695.

[39] S. Berghofer, M. Strecker, Extracting a formally verified, fully executable compiler from a proof assistant, in: Proceedings of the 2nd International Workshop on Compiler Optimization Meets Compiler, COCV'03, ENTCS 82 (2), Elsevier, 2004, pp. 377–394.

[40] S. Blazy, Z. Dargaye, X. Leroy, Formal verification of a C compiler front-end, in: Proceedings of the 14th Symposium on Formal Methods, FM'06, LNCS 4085, Springer, 2006, pp. 460–475.

[41] X. Leroy, Formal certification of a compiler back-end, or: programming a compiler with a proof assistant, in: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'06, ACM Press, 2006, pp. 42–54.

[42] X. Leroy, A formally verified compiler back-end, Journal of Automated Reasoning 43 (4), Springer (2009) 363–446.

[43] C. Lattner, V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in: Proceedings of the 2004 International Symposium on Code Generation and Optimization, CGO'04, ACM, 2004, pp. 75–87.

Appendix A. Proofs

Lemma 27. *For all $S, S', h, h', td, k_0, k, e, v, \rho, cs, cs'$ of their respective types, such that*

$$\begin{array}{ll} td = topDepth(\rho) & (is, cs) = trE\ e\ \rho\ F \\ S' = drop\ td\ S & cs' \supseteq cs \\ k_0 \leq k & \end{array}$$

If $\underbrace{(is, h, k_0, k, S, cs')}_{c_{init}} \rightarrow^*_{S'} \underbrace{([\texttt{POPCONT}], h', k', k', v : S', cs')}_{c_{final}}$ *then* $k' = k_0$.

Proof. This can be trivially proved by induction on the length of $\rightarrow^*_{S'}$ and by cases over the expression e whose translation is executed. □

Theorem 26. *For all $S, S', E, h, h', td, k_0, k, e, v, \delta, m, s, \rho, cs, cs', F,$ and Σ of their respective types, if*

$$\begin{array}{lll} E\ admissible\ w.r.t.\ k & (F, cs')\ generated\ from\ \Sigma & \\ E \equiv (\rho, S) & (is, cs) = trE\ e\ \rho\ F & td = topDepth(\rho) \\ S' = drop\ td\ S & cs' \supseteq cs & k_0 \leq k \end{array}$$

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

1. $\underbrace{(is, h, k_0, k, S, cs')}_{c_{init}} \rightarrow^*_{S'} \underbrace{([\texttt{POPCONT}], h' \mid_{k_0}, k_0, k_0, v : S', cs')}_{c_{final}}$
2. $\delta = diff(k, h, h')$
3. $m = maxFreshCells(c_{init} \rightarrow^*_{S'} c_{final})$
4. $s = maxFreshWords(c_{init} \rightarrow^*_{S'} c_{final})$

Proof. The ($\Rightarrow$) direction is shown by induction on the $\Downarrow$-derivation. We distinguish cases depending on the last $\Downarrow$ rule applied.

- **Case** [*Lit*]: $e \equiv c$

  Let $(is, cs) = trE\ e\ \rho\ F$ be the SVM code generated, where $cs = [\,]$ and:

$$is = \texttt{BUILDENV}\ [c] : \overbrace{\texttt{SLIDE}\ 1\ (\mathit{topDepth}(\rho)) : \underbrace{\texttt{DECREGION} : \underbrace{[\texttt{POPCONT}]}_{is_3}}_{is_2}}^{is_1}$$

By the rule $[Lit]$ we have $E \vdash (h,k), td, c \Downarrow (h,k), c, ([\,]_k, 0, 1)$. For all cs' we get the following derivation:

$$\begin{array}{ll}
 & \overbrace{(\texttt{BUILDENV}\ [c] : is_1,\quad h,\quad k_0,\quad k,\quad S,\quad cs')}^{c_{init}} \\
\rightarrow_{S'} & \\
 & (\texttt{SLIDE}\ 1\ (\mathit{topDepth}(\rho)) : is_2,\quad h,\quad k_0,\quad k,\quad c : S,\quad cs') \\
\rightarrow_{S'} & \qquad \{\text{since } td = \mathit{topDepth}(\rho)\} \\
 & (\texttt{DECREGION} : is_3,\quad h,\quad k_0,\quad k,\quad c : \mathit{drop}\ td\ S,\quad cs') \\
\rightarrow_{S'} & \\
 & \underbrace{([\texttt{POPCONT}],\quad h|_{k_0},\quad k_0,\quad k_0,\quad c : \mathit{drop}\ td\ S,\quad cs')}_{c_{final}}
\end{array}$$

Because of the assumption $S' = \mathit{drop}\ td\ S$ the property (1) holds. Moreover, we get $\mathit{diff}(k,h,h) = [i \mapsto 0 \mid 0 \le i \le k] = \delta$, so property (2) holds. Properties (3) and (4) hold because $\mathit{maxFreshCells}(c_{init} \rightarrow^*_{S'} c_{final}) = 0$ and $\mathit{maxFreshWords}(c_{init} \rightarrow^*_{S'} c_{final}) = 1$.

- **Case** $[Var]$: $e \equiv x$

  The proof is similar to the case $e \equiv c$. Now $E[x \mapsto v] \vdash (h,k), td, x \Downarrow (h,k), v, ([\,]_k, 0, 1)$ holds and the code generated is as follows:

$$is = \texttt{BUILDENV}\ [\rho(x)] : \overbrace{\texttt{SLIDE}\ 1\ (\mathit{topDepth}(\rho)) : \underbrace{\texttt{DECREGION} : \underbrace{[\texttt{POPCONT}]}_{is_3}}_{is_2}}^{is_1}$$

  From which the following derivation is obtained:

$$
\begin{array}{ll}
 & \overbrace{(\texttt{BUILDENV}\ [\rho(x)] : is_1,\ \ h,\ \ k_0,\ \ k,\ \ S,\ \ cs')}^{c_{init}} \\
\rightarrow_{S'} & \quad \{\text{since } S!(\rho(x)) = E(x) = v\} \\
 & (\texttt{SLIDE}\ 1\ (topDepth(\rho)) : is_2,\ \ h,\ \ k_0,\ \ k,\ \ v : S,\ \ cs') \\
\rightarrow_{S'} & \quad \{\text{since } td = topDepth(\rho)\} \\
 & (\texttt{DECREGION} : is_3,\ \ h,\ \ k_0,\ \ k,\ \ v : drop\ td\ S,\ \ cs') \\
\rightarrow_{S'} & \\
 & \underbrace{([\texttt{POPCONT}],\ \ h|_{k_0},\ \ k_0,\ \ k_0,\ \ v : drop\ td\ S,\ \ cs')}_{c_{final}}
\end{array}
$$

Hence property (1) holds. The proof of (2), (3) and (4) is analogous to the case $e \equiv c$.

- **Case** $[Copy]$: $e \equiv x\ @\ r$

  On the one hand, by using the $[Copy]$ rule we know that:

$$E \vdash (h,k), td, x\ @\ r \Downarrow (h',k), p', ([j \mapsto size(h,p)]_k, size(h,p), 2)$$

  where $E(x) = p$, $E(r) = j \leq k$ and $(h',p') = copy(h,p,j)$. On the other hand, we obtain $(is, [\,]) = trE\ e\ \rho\ F$ where:

$$
is = \texttt{BUILDENV}\ [\rho(x), \rho(r)] : \overbrace{\texttt{COPY} : \overbrace{\texttt{SLIDE}\ 1\ (topDepth(\rho)) : \underbrace{\texttt{DECREGION} : \underbrace{[\texttt{POPCONT}]}_{is_4}}_{is_3}}^{is_2}}^{is_1}
$$

  For any cs', the first step of the $\rightarrow^*_{S'}$ derivation is as follows:

$$
\begin{array}{ll}
 & \overbrace{(\texttt{BUILDENV}\ [\rho(x), \rho(r)] : is_1,\ \ h,\ \ k_0,\ \ k,\ \ S,\ \ cs')}^{c_{init}} \\
\rightarrow_{S'} & \quad \{\ \text{since } E(x) = p = S!(\rho(x))\ \} \\
 & (\texttt{COPY} : is_2,\ \ h,\ \ k_0,\ \ k,\ \ p : Item_k(\rho(r)) : S,\ \ cs')
\end{array}
$$

Let us proceed by case distinction: on one hand, if $r \neq \mathit{self}$ then $\mathit{Item}_k(\rho(r)) = S!(\rho(r)) = E(r) = j$. On the other hand, if $r = \mathit{self}$ then we have $\mathit{Item}_k(\rho(r)) = \mathit{Item}_k(\mathit{self}) = k$, but since $E(\mathit{self}) = k$ (by Proposition 15) and $E(\mathit{self}) = j$ (by rule $[\mathit{Copy}]$), we have that $\mathit{Item}_k(\rho(r)) = j$). Hence the current machine configuration can be rewritten as $(\texttt{COPY} : is_2, h, k_0, k, p : j : S, cs')$.

We shall now resume the $\rightarrow^*_{S'}$ derivation:

$$
\begin{array}{ll}
 & (\texttt{COPY} : is_2, \quad h, \quad k_0, \quad k, \quad p : j : S, \quad cs') \\
\rightarrow_{S'} & \quad \{ \text{ since } (h', p') = \mathit{copy}(h, p, j) \text{ and } j \leq k\} \\
 & (\texttt{SLIDE } 1 \ (\mathit{topDepth}(\rho)) : is_3, \quad h', \quad k_0, \quad k, \quad p' : S, \quad cs') \\
\rightarrow_{S'} & \quad \{\text{since } td = \mathit{topDepth}(\rho)\} \\
 & (\texttt{DECREGION} : is_4, \quad h', \quad k_0, \quad k, \quad p' : \mathit{drop}\ td\ S, \quad cs') \\
\rightarrow_{S'} & \\
 & \underbrace{([\texttt{POPCONT}], \quad h'|_{k_0}, \quad k_0, \quad k_0, \quad p' : \mathit{drop}\ td\ S, \quad cs')}_{c_{final}}
\end{array}
$$

Therefore (1) holds, because of the assumption $S' = \mathit{drop}\ td\ S$. Properties (2) and (3) follow from Lemma 25. With respect to (4), from the resulting $\rightarrow^*_{S'}$ derivation it can be easily shown that $\mathit{maxFreshWords}(c_{init} \rightarrow^*_{S'} c_{final}) = 2$.

- **Case** $[\mathit{PrimOp}]$: $e \equiv a_1 \oplus a_2$

  The translation yields the following instruction sequence:

$$
is = \texttt{BUILDENV } [\rho(a_1), \rho(a_2)] : \overbrace{\texttt{PRIMOP} \oplus : \overbrace{\texttt{SLIDE } 1 \ (\mathit{topDepth}(\rho)) : \underbrace{\texttt{DECREGION} : \underbrace{[\texttt{POPCONT}]}_{is_4}}_{is_3}}^{is_2}}^{is_1}
$$

  By executing it, we obtain the following derivation:

$$
\begin{array}{ll}
 & \overbrace{(\texttt{BUILDENV } [\rho(a_1), \rho(a_2)] : is_1, \quad h, \quad k_0, \quad k, \quad S, \quad cs')}^{c_{init}} \\
\rightarrow_{S'} & \quad \{\text{since } S!(\rho(a_i)) = E(a_i) \text{ for every } i \in \{1..2\}, \text{ see below } \} \\
 & (\texttt{PRIMOP} \oplus : is_2, \quad h, \quad k_0, \quad k, \quad E(a_1) : E(a_2) : S, \quad cs')
\end{array}
$$

$\rightarrow_{S'}$
$(\texttt{SLIDE}\ 1\ (\mathit{topDepth}(\rho)) : is_3,\ \ h,\ \ k_0,\ \ k,\ \ (E(a_1) \oplus E(a_2)) : S,\ \ cs')$
$\rightarrow_{S'}$
$(\texttt{DECREGION} : is_4,\ \ h,\ \ k_0,\ \ k,\ \ (E(a_1) \oplus E(a_2)) : \mathit{drop}\ td\ S,\ \ cs')$
$\rightarrow_{S'}$
$([\texttt{POPCONT}],\ \ h|_{k_0},\ \ k_0,\ \ k_0,\ \ (E(a_1) \oplus E(a_2)) : \mathit{drop}\ td\ S,\ \ cs') \equiv c_{\mathit{final}}$

Hence, property 1 holds. From the initial and final configurations we obtain $\mathit{diff}(k, h, h) = [i \mapsto 0 \mid 0 \leq i \leq k] = \delta$, so property (2) holds. Properties (3) and (4) hold because $\mathit{maxFreshCells}(c_{\mathit{init}} \rightarrow^*_{S'} c_{\mathit{final}}) = 0$ and $\mathit{maxFreshWords}(c_{\mathit{init}} \rightarrow^*_{S'} c_{\mathit{final}}) = 2$.

- **Case** $[\mathit{Cons}]$: $e \equiv C\ \overline{a_i}^n\ @\ r$

  In the big-step semantics $E \vdash (h, k), e \Downarrow (h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)], k), p, ([E(r) \mapsto 1]_k, 1, 1)$ where p is a fresh pointer. The translation yields the following instruction sequence:

$$is = \texttt{BUILDCLS}\ C\ [\overline{\rho(a_i)}^n]\ \rho(r) : \overbrace{\texttt{SLIDE}\ 1\ (\mathit{topDepth}(\rho)) : \underbrace{\texttt{DECREGION} : \underbrace{[\texttt{POPCONT}]}_{is_3}}_{is_2}}^{is_1}$$

  By executing this, we obtain the following derivation:

$$\begin{array}{ll}
 & \overbrace{(\texttt{BUILDCLS}\ C\ [\overline{\rho(a_i)}^n]\ \rho(r) : is_1,\ \ h,\ \ k_0,\ \ k,\ \ S,\ \ cs')}^{c_{\mathit{init}}} \\
\rightarrow_{S'} & \quad \{\text{since } S!(\rho(a_i)) = E(a_i) \text{ for every } i \in \{1..n\}, \text{ see below }\} \\
 & (\texttt{SLIDE}\ 1\ (\mathit{topDepth}(\rho)) : is_2,\ \ h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)], \\
 & \ \ k_0,\ \ k,\ \ p : S,\ \ cs') \\
\rightarrow_{S'} & \quad \{\text{since } td = \mathit{topDepth}(\rho)\} \\
 & (\texttt{DECREGION} : is_3,\ \ h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)], \\
 & \ \ k_0,\ \ k,\ \ p : \mathit{drop}\ td\ S,\ \ cs') \\
\rightarrow_{S'} & \\
 & ([\texttt{POPCONT}],\ \ (h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)])|_{k_0}, \\
 & \ \ k_0,\ \ k_0,\ \ p : \mathit{drop}\ td\ S,\ \ cs') \equiv c_{\mathit{final}}
\end{array}$$

In the first step we have assumed that $Item_k(\rho(r)) = E(r)$. The proof of this is similar to that seen in the $[Copy]$ rule. Thus (1) holds. From the initial and final configurations we obtain:

$$diff(k, h, h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^{\,n})]) = [E(r) \mapsto 1]_k$$

$$maxFreshCells(c_{init} \rightarrow^*_{S'} c_{final}) = 1$$

$$maxFreshWords(c_{init} \rightarrow^*_{S'} c_{final}) = 1$$

from which (2), (3) and (4) follow trivially.

- **Case** $[App]$: $e \equiv g\ \overline{a_i}^{\,n}\ @\ \overline{r_j}^{\,l}$

  We assume, by rule $[App]$ that $E_g \vdash (h, k+1), n+l, e_g \Downarrow (h', k+1), v, (\delta, m, s)$, where E_g is defined as follows:

$$E_g = [\overline{y_i \mapsto E(a_i)}^{\,n}, \overline{r'_j \mapsto E(r_j)}^{\,l}, self \mapsto k+1]$$

  and $(g\ \overline{y_i}^{\,n}\ @\ \overline{r'_j}^{\,l} = e_g) \in \Sigma$. Moreover, there is a code pointer $\mathbf{p}$ such that the binding $[g \mapsto \mathbf{p}]$ belongs to F, as specified by the trE function. Since (F, cs') is generated from Σ (by assumption), we know (by Definition 17) that there exist some n', l', $\overline{x'_i}^{\,n'}$, $\overline{r_j}^{\,l'}$, e'_g e'_g, $cs_g \subseteq cs'$, and $F' \subseteq F$ such that the following conditions hold:

$$(g\ \overline{x'_i}^{\,n'}\ @\ \overline{r'_j}^{\,m'} = e'_g) \in \Sigma$$

$$trF\ (g\ \overline{x'_i}^{\,n'}\ @\ \overline{r'_j}^{\,m'} = e'_g)\ F' = (cs_g, F' \uplus [g \mapsto \mathbf{p}])$$

  However, a function definition can only appear once in the Σ environment, hence we know that $n = n'$, $l = l'$, $\overline{x_i}^{\,n} = \overline{x'_i}^{\,n'}$, $\overline{r_j}^{\,l} = \overline{r'_j}^{\,l'}$, and $e_g = e'_g$. Hence we can rewrite the last equation above as follows:

$$trF\ (g\ \overline{x_i}^{\,n}\ @\ \overline{r_j}^{\,m} = e_g)\ F' = (cs_g, F' \uplus [g \mapsto \mathbf{p}])$$

  which implies, by the definition of trF, the existence of an instruction sequence is_g and a code store cs'_g such that

$$(is_g, cs'_g) = trE\ e_g\ \rho_g\ F'$$

  where $cs_g = cs'_g \uplus [\mathbf{p} \mapsto is_g]$, and ρ_g is the following:

$$\rho_g = [(\overline{[r'_j \mapsto (l-j)+1}^{\,l}, \overline{y_i \mapsto (n-i)+(l+1)}^{\,n}], n+l, 0)]$$

Since $cs_g \subseteq cs'$, we know that $cs(\mathbf{p}) = is_g$. Now, let us translate the function application: we obtain $(is, cs) = trE\ e\ \rho\ F$, where:

$$is = \texttt{BUILDENV}\ [\overline{\rho(a_i)}^{n}, \overline{\rho(r_j)}^{l}] : \overbrace{\texttt{SLIDE}\ (n+l)\ (topDepth(\rho)) : \underbrace{[\texttt{CALL}\ \mathbf{p}]}_{is_2}}^{is_1}$$

For any cs' such that $cs' \supseteq cs$, the code generated leads to the following SVM derivation:

$$\begin{array}{ll}
 & \overbrace{(\texttt{BUILDENV}\ [\overline{\rho(a_i)}^{n}, \overline{\rho(r_j)}^{l}] : is_1,\quad h,\quad k_0,\quad k,\quad S,\quad cs')}^{c_{init}} \\
\rightarrow_{S'} & \\
 & (\texttt{SLIDE}\ (n+l)\ (topDepth(\rho)) : is_2,\quad h,\quad k_0,\quad k, \\
 & \qquad\qquad \overline{S!(\rho(a_i))}^{n} : \overline{S!(\rho(r_j))}^{l} : S,\quad cs') \\
\rightarrow_{S'} & \quad \{\ \text{since } td = topDepth(\rho)\ \} \\
 & ([\texttt{CALL}\ \mathbf{p}],\quad h,\quad k_0,\quad k,\quad \overline{S!(\rho\ a_i)}^{n} : \overline{S!(\rho\ r'_j)}^{l} : drop\ td\ S,\quad cs') \\
\rightarrow_{S'} & \quad \{\ \text{since } cs'(\mathbf{p}) = cs(\mathbf{p}) = is_g\ \} \\
 & (is_g,\quad h,\quad k_0,\quad k+1,\quad \overline{S!(\rho\ a_i)}^{n} : \overline{S!(\rho\ r'_j)}^{l} : drop\ td\ S,\quad cs') \equiv c_f \\
\rightarrow^{*}_{S'} & \quad \{\ \text{by i.h. (see below) and } k_0 \leq k\ \} \\
 & \underbrace{([\texttt{POPCONT}],\quad h'|_{k_0},\quad k_0,\quad k_0,\quad v : drop\ td\ S,\quad cs')}_{c_{final}}
\end{array}$$

where $S' = drop\ td\ S$. In order to apply the induction hypothesis in the last step, we prove that every assumption of the theorem holds for the $\Downarrow$ derivation corresponding to e_f. We know that, if (F, cs') is generated from Σ, then so is (F', cs'), since $F' \subseteq F$. The only nontrivial assumption left to prove is that $E_g \equiv (\rho_g, S_g)$, denoting by S_g the result from pushing the actual parameters $\overline{S!(\rho\ a_i)}^{n}$ and $\overline{S!(\rho\ r'_j)}^{l}$ onto S. Firstly, let $x \in \operatorname{dom} \rho_g$. We prove that $S_g!(\rho_g(x)) = E_g(x)$. On the one hand, if $x = y_i$ for some $i \in \{1..n\}$, we get $\rho_g(x) = n + l - [(n - i) + (l + 1)] = i - 1$. Hence:

$$S_g!(\rho_g(x)) = S_g!(i-1) = S!(\rho(a_i)) = E(a_i) = E_g(y_i)$$

On the other hand, if $x = r'_j$ for some $j \in \{1..l\}$ then we get $\rho_g(x) = n + l - [(l - j) + 1] = n + j - 1$ and:

$$S_g!(\rho_g(x)) = S_g!(n + j - 1) = S!(\rho(r_j)) = E(r_j) = E_g(r'_j)$$

So $E_g \equiv (\rho_g, S_g)$, and the induction hypothesis can be applied on the derivation of e_g, which proves property (1) since $\mathit{drop}\ (n + l)\ S_g = \mathit{drop}\ td\ S$.

The proof for (2) can be easily established from the fact that $\mathit{diff}(k, h, h') = \mathit{diff}(k + 1, h, h')|_k$ and that $\delta = \mathit{diff}(k + 1, h, h')$ by induction hypothesis. Property (3) also follows trivially from the induction hypothesis. With respect to (4), let us denote:

$$\begin{array}{lcl} S_1 & = & \overline{S!(\rho(a_i))}^n : \overline{S!(\rho(r_j))}^l : S \\ S_2 & = & \overline{S!(\rho(a_i))}^n : \overline{S!(\rho(r_j))}^l : \mathit{drop}\ td\ S \\ S_3 & = & v : \mathit{drop}\ td\ S \end{array}$$

$$\begin{array}{rl} \mathit{maxFreshWords}(c_{\mathit{init}} \rightarrow^*_{S'} c_f) = & \max\{\mathit{sizeST}(S_1) - \mathit{sizeST}(S), \\ & \quad \mathit{sizeST}(S_2) - \mathit{sizeST}(S)\} \\ = & \{n + l, n + l - td\} \\ = & n + l \end{array}$$

On the other hand, $\mathit{maxFreshWords}(c_f \rightarrow^*_{S'} c_{\mathit{final}}) = s$ by induction hypothesis. Therefore:

$$\begin{array}{l} \mathit{maxFreshWords}(c_{\mathit{init}} \rightarrow^*_{S'} c_f \rightarrow^*_{S'} c_{\mathit{final}}) \\ \quad = \max\{\mathit{maxFreshWords}(c_{\mathit{init}} \rightarrow^*_{S'} c_f), \\ \qquad \mathit{maxFreshWords}(c_f \rightarrow^*_{S'} c_{\mathit{final}}) \\ \qquad + \mathit{sizeST}(S_2) - \mathit{sizeST}(S)\} \\ \quad = \max\{n + l, s + n + l - td\} \end{array}$$

- **Case** [*Let*]: $e \equiv \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2$

  By rule [*Let*] we get:

$$\begin{array}{l} E \vdash (h, k), 0, e_1 \Downarrow (h', k), v_1, (\delta_1, m_1, s_1) \\ E_1 \vdash (h', k), td + 1, e_2 \Downarrow (h'', k), v, (\delta_2, m_2, s_2) \end{array}$$

where $E_1 = E \uplus [x_1 \mapsto v_1]$. In addition, we assume $(is_1, cs_1) = trE\ e_1\ \rho^{+\!\!+}\ F$ and $(is_2, cs_2) = trE\ e_2\ \rho_1\ F$ where $\rho_1 = \rho + [x_1 \mapsto 1]$. On the other hand, we get $cs \supseteq cs_1$, $cs \supseteq cs_2$ and $is = \texttt{PUSHCONT}\ \mathbf{p} : is_1$, in which $cs_1(\mathbf{p}) = is_2$. Given $cs' \supseteq cs$, the trace corresponding to the execution of e starts as follows:

$$
\begin{array}{ll}
 & \overbrace{(\texttt{PUSHCONT}\ \mathbf{p} : is_1,\quad h,\quad k_0,\quad k,\quad S,\quad cs')}^{c_{init}} \\
\rightarrow_{S'} & \\
 & (is_1,\quad h,\quad k,\quad k,\quad (k_0, \mathbf{p}) : S,\quad cs') \equiv c_1
\end{array}
$$

In order to apply induction hypothesis on e_1 we have to check that $E \equiv (\rho^{+\!\!+}, (k_0, \mathbf{p}) : S)$, which trivially holds from the hypothesis $E \equiv (\rho, S)$ and because dom ρ = dom $\rho^{+\!\!+}$. The rest of the assumptions hold trivially.

In this case $td = 0$, so by applying i.h. we obtain:

$$
\begin{array}{ll}
 & c_1 \\
\rightarrow^*_{(k_0,\mathbf{p}):S} & \\
 & ([\texttt{POPCONT}],\ h'|_k,\ k,\ k,\ v_1 : drop\ 0\ ((k_0, \mathbf{p}) : S),\ cs'[\mathbf{p} \mapsto is_2]) \equiv c_2
\end{array}
$$

which trivially implies $c_1 \rightarrow^*_{S'} c_2$.

Consequently:

$$
\begin{array}{ll}
 & \overbrace{(\texttt{PUSHCONT}\ \mathbf{p} : is_1,\quad h,\quad k_0,\quad k,\quad S,\quad cs')}^{c_{init}} \\
\rightarrow_{S'} & \\
 & (is_1,\quad h,\quad k,\quad k,\quad (k_0, \mathbf{p}) : S,\quad cs') \\
\rightarrow^*_{S'} & \\
 & ([\texttt{POPCONT}],\quad h'|_k,\ k,\ k,\ v_1 : drop\ 0\ ((k_0, \mathbf{p}) : S),\ cs'[\mathbf{p} \mapsto is_2]) \equiv c_2 \\
\rightarrow_{S'} & \\
 & (is_2,\quad h'|_k,\quad k_0,\quad k,\quad v_1 : S,\quad cs') \equiv c_3 \\
\rightarrow^*_{S'} & \quad \{\text{ by i.h. (see below) }\} \\
 & ([\texttt{POPCONT}],\quad h''|_{k_0},\quad k_0,\quad k_0,\quad v : drop\ (td+1)\ (v_1 : S),\quad cs') \\
\equiv & \\
 & \underbrace{([\texttt{POPCONT}],\quad h''|_{k_0},\quad k_0,\quad k_0,\quad v : S',\quad cs')}_{c_{final}}
\end{array}
$$

In order to apply the induction hypothesis on e_2 in the last step we have to check that $E_1 \equiv (\rho_1, v_1 : S)$. Let $x \in \mathrm{dom}(E_1)$: if $x = x_1$ then $(v_1 : S)!(\rho_1(x)) = (v_1 : S)!0 = v_1 = E_1(x)$. If $x \neq x_1$ then $(v_1 : S)!(\rho_1(x)) = (v_1 : S)!(\rho(x) + 1) = S!(\rho(x)) = E(x) = E_1(x)$. Additionally $\mathit{drop}\ (td + 1)\ (v_1 : S) = \mathit{drop}\ td\ S = S'$. The rest of the assumptions hold trivially.

Hence (1) holds. With respect to (2), let $i \in \{1..k\}$

$$
\begin{aligned}
&(\mathit{diff}\,(k, h, h''))(i)\\
&\quad = |\{p \in h'' \mid \mathit{region}(h''(p)) = i\}| - |\{p \in h \mid \mathit{region}(h(p)) = i\}|\\
&\quad = |\{p \in h'' \mid \mathit{region}(h''(p)) = i\}| - |\{p \in h' \mid \mathit{region}(h'(p)) = i\}|\\
&\qquad + |\{p \in h' \mid \mathit{region}(h'(p)) = i\}| - |\{p \in h \mid \mathit{region}(h(p)) = i\}|\\
&\quad = (\mathit{diff}\,(k, h, h'))(i) + (\mathit{diff}\,(k, h', h''))(i)\\
&\quad = \delta_1(i) + \delta_2(i)
\end{aligned}
$$

Therefore $\mathit{diff}\,(k, h, h'') = \delta_1 + \delta_2$. Properties (3) and (4) are proven as follows:

$$
\begin{aligned}
&\mathit{maxFreshCells}(c_{\mathit{init}} \to^*_{S'} c_2) \quad = m_1\\
&\mathit{maxFreshCells}(c_2 \to^*_{S'} c_{\mathit{final}}) \quad = m_2\\
&\mathit{maxFreshCells}(c_{\mathit{init}} \to^*_{S'} c_{\mathit{final}})\\
&\quad = \max\{m_1, m_2 + \mathit{sizeH}(c_2) - \mathit{sizeH}(c_{\mathit{init}})\}\\
&\quad = \max\{m_1, m_2 + \textstyle\sum_{i=0}^{k}(\mathit{diff}\,(k, h, h'))(i)\}\\
&\quad = \max\{m_1, m_2 + |\delta_1|\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{maxFreshWords}(c_{\mathit{init}} \to^*_{S'} c_1) && = 2\\
&\mathit{maxFreshWords}(c_{\mathit{init}} \to^*_{S'} c_2) && = \max\ \{2, 2 + s_1\} = 2 + s_1\\
&\mathit{maxFreshWords}(c_{\mathit{init}} \to^*_{S'} c_2 \to^*_{S'} c_3) && = \max\ \{2 + s_1, 1\} = 2 + s_1\\
&\mathit{maxFreshWords}(c_{\mathit{init}} \to^*_{S'} c_3 \to^*_{S'} c_{\mathit{final}}) && = \max\ \{2 + s_1, 1 + s_2\}
\end{aligned}
$$

- **Case** [*Case*]: $e \equiv \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$

  We assume that the r-th **case** alternative is executed under an environment $E_r \stackrel{\text{def}}{=} E \uplus [\overline{x_{ri} \mapsto v_i}^{n_r}]$, where the values v_i are the parameters of the data construction pointed to by $E(x)$.

$$E_r \vdash (h, k), td + n_r, e_r \Downarrow (h', k), v, (\delta, m, s)$$

In addition, let us denote $\rho_r \stackrel{\text{def}}{=} \rho + [\overline{x_{ri} \mapsto n_r - i - 1}^{n_r}]$. If $(is_r, cs_r) = trE\ e_r\ \rho_r\ F$ then it holds that $cs \supseteq cs_r$. Moreover, we get $is = \texttt{MATCH}\ (\rho(x))\ \ \overline{\mathbf{p}_j}^m$ with $cs(\mathbf{p}_r) = is_r$ for each $r \in \{1..n\}$. The SVM derivation corresponding to is results as follows:

$$
\begin{array}{ll}
 & \overbrace{(\texttt{MATCH}\ (\rho(x))\ \overline{\mathbf{p}_j}^m,\ \ h[p \mapsto (j, C_r\ \overline{v_i}^{n_r})],\ \ k_0,\ \ k,\ \ S,\ \ cs'[\mathbf{p}_r \mapsto is_r])}^{c_{init}} \\
\rightarrow_{S'} & \quad \{\ \text{since } S!(\rho(x)) = E(x) = p\ \} \\
 & (is_r,\ \ h,\ \ k_0,\ \ k,\ \ \overline{v_i}^{n_r} : S,\ \ cs') \equiv c_1 \\
\rightarrow^*_{S'} & \quad \{\ \text{by i.h. (see below)}\ \} \\
 & ([\texttt{POPCONT}],\ \ h|_{k_0},\ \ k_0,\ \ k_0,\ \ v : drop\ (td + n_r)\ (\overline{v_i}^{n_r} : S),\ \ cs') \\
\equiv & \\
 & \underbrace{([\texttt{POPCONT}],\ \ h|_{k_0},\ \ k_0,\ \ k_0,\ \ v : drop\ td\ S,\ \ cs')}_{c_{final}}
\end{array}
$$

In the same way as in the previous cases, we have to ensure that $E_r \equiv (\rho_r, \overline{v_i}^{n_r} : S)$. Let $z \in \text{dom}\ \rho_r$. We made the following case distinction: on one hand, if $z = x_{ri}$ for some $i \in \{1..n_r\}$ then $\rho_r(z) = i - 1$ and hence:

$$(\overline{v_{ir}}^n : S)!(\rho_r(z)) = (\overline{v_{ir}}^n : S)!(i-1) = v_i = E_r(x_{ri}) = E_r(z)$$

On the other hand, if $z \neq x_{ri}$ for all $i \in \{1..n_r\}$ then $\rho_r(z) = \rho(x) + n_r$. Therefore:

$$(\overline{v_{ir}}^n : S)!(\rho_r(z)) = (\overline{v_{ir}}^n : S)!(\rho(x) + n_r) = S!(\rho(x)) = E(z) = E_r(z)$$

Hence we get $E_r \equiv (\rho_r, \overline{v_i}^{n_r} : S)$ and (1) holds by applying i.h. since $drop\ (td + n_r)\ S_g = drop\ td\ S$.

Properties (2) and (3) follow trivially from the induction hypothesis. With respect to (4) we get:

$$
\begin{array}{ll}
maxFreshWords(c_{init} \rightarrow^*_{S'} c_1) & = n_r \\
maxFreshWords(c_1 \rightarrow^*_{S'} c_{final}) & = s \\
maxFreshWords(c_{init} \rightarrow^*_{S'} c_{final}) & = \max\ \{n_r, s + n_r\} = s + n_r
\end{array}
$$

- **Case** [*Case!*]: $e \equiv \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$

  The proofs of (1) and (4) are similar to those seen for the nondestructive **case**. Property (2) follows trivially from the induction hypothesis and the definition of *diff*. With respect to (3), let c_1 denote the SVM state prior to the execution of the branch e_r. Then:

  $$\begin{array}{ll} \mathit{maxFreshCells}(c_{\mathit{init}} \to^*_{S'} c_1) & = 0 \\ \mathit{maxFreshCells}(c_1 \to^*_{S'} c_{\mathit{final}}) & = m \\ \mathit{maxFreshCells}(c_{\mathit{init}} \to^*_{S'} c_{\mathit{final}}) & = \max\ \{0, m-1\} \end{array}$$

  which proves the desired result.

The ($\Leftarrow$) direction of the theorem can be proved by induction on the length of the $\to^*_{S'}$ derivation. Since this proof is mostly standard, we only describe it briefly: since it is known that the SVM is deterministic, we prove that, given a *Safe* program, if the SVM eventually halts then a corresponding $\Downarrow$-derivation can be built. We distinguish cases depending on the expression e being evaluated. The base cases $e \equiv c$, $e \equiv x$, $e \equiv x\ @\ r$, $e \equiv a_1 \oplus a_2$, and $e \equiv C\ \overline{a_i}^n\ @\ r$ are straightforward: if the SVM machine halts with the corresponding result on the top of stack, this result can be used with rules [*Lit*], [*Var*], [*Copy*] and [*Cons*].

As an example, let us consider $e \equiv x$. Let $(is, cs) = \mathit{trE}\ e\ \rho\ F$ be the SVM code generated, where $cs = [\,]$ and:

$$is = \texttt{BUILDENV}\ [\rho(x)] : \overbrace{\texttt{SLIDE}\ 1\ (\mathit{topDepth}(\rho)) : \underbrace{\texttt{DECREGION} : \underbrace{[\texttt{POPCONT}]}_{is_3}}_{is_2}}^{is_1}$$

Given any cs', the only possible derivation is the following one:

$$\begin{array}{ll} & \overbrace{(\texttt{BUILDENV}\ [\rho(x)] : is_1,\ \ h,\ \ k_0,\ \ k,\ \ S,\ \ cs')}^{c_{\mathit{init}}} \\ \to_{S'} & \quad \{\text{since } S!(\rho(x)) = E(x) = v\} \\ & (\texttt{SLIDE}\ 1\ (\mathit{topDepth}(\rho)) : is_2,\ \ h,\ \ k_0,\ \ k,\ \ v : S,\ \ cs') \\ \to_{S'} & \quad \{\text{since } td = \mathit{topDepth}(\rho)\} \end{array}$$

$$
\begin{array}{ll}
 & (\texttt{DECREGION} : is_3,\ \ h,\ \ k_0,\ \ k,\ \ v : drop\ td\ S,\ \ cs') \\
\rightarrow_{S'} & \\
 & \underbrace{([\texttt{POPCONT}],\ \ h|_{k_0},\ \ k_0,\ \ k_0,\ \ v : drop\ td\ S,\ \ cs')}_{c_{final}}
\end{array}
$$

Trivially, by rule $[Var]$, $E[x \mapsto v] \vdash (h,k), td, x \Downarrow (h,k), v, ([\ ]_k, 0, 1)$. Property (2) holds because $diff(k,h,h) = [i \mapsto 0 \mid 0 \leq i \leq k] = \delta$. Properties (3) and (4) hold because $maxFreshCells(c_{init} \rightarrow^*_{S'} c_{final}) = 0$ and $maxFreshWords\ (c_{init} \rightarrow^*_{S'} c_{final}) = 1$.

With respect to the remaining cases, the $\rightarrow^*_{S'}$ execution sequence is made up of a set of preliminary actions (building the variable environment in the case of function application, pushing a continuation in the case of **let** expression or executing a MATCH/MATCH! at the beginning of a **case**/**case**!) followed by the evaluation of the corresponding subexpressions (i.e, either the function being called or the main/auxiliary expressions of a **let** or the **case**/**case**! branch). The induction hypothesis can be applied to these $(\rightarrow^*)$-subderivations in order to get the required assumptions of the corresponding $\Downarrow$ rule.

As an example, consider $e \equiv$ **let** $x_1 = e_1$ **in** e_2. Assume $(is_1, cs_1) = trE\ e_1\ \rho^{+\!\!+}\ F$ and $(is_2, cs_2) = trE\ e_2\ \rho_1\ F$ where $\rho_1 = \rho + [x_1 \mapsto 1]$. Then $is = \texttt{PUSHCONT}\ \mathbf{p} : is_1$, in which $cs_1(\mathbf{p}) = is_2$. Given $cs' \supseteq cs$, the trace corresponding to the execution of e must be as follows:

$$
\begin{array}{ll}
 & \overbrace{(\texttt{PUSHCONT}\ \mathbf{p} : is_1,\ \ h,\ \ k_0,\ \ k,\ \ S,\ \ cs')}^{c_{init}} \\
\rightarrow_{S'} & \\
 & (is_1,\ \ h,\ \ k,\ \ k,\ \ (k_0, \mathbf{p}) : S,\ \ cs') \equiv c_1 \\
\rightarrow^*_{S'} & \{\text{by Lemma 27}\} \\
 & ([\texttt{POPCONT}],\ \ h',\ k,\ k,\ v_1 : (k_0, \mathbf{p}) : S),\ cs'[\mathbf{p} \mapsto is_2]) \equiv c_2 \\
\rightarrow_{S'} & \\
 & (is_2,\ \ h',\ \ k_0,\ \ k,\ \ v_1 : S,\ \ cs') \equiv c_3 \\
\rightarrow^*_{S'} & \\
 & \underbrace{([\texttt{POPCONT}],\ \ h'',\ \ k_0,\ \ k_0,\ \ v : S',\ \ cs')}_{c_{final}}
\end{array}
$$

Notice that c_1 contains a continuation $(k_0, \mathbf{p})$ on top of the stack. The only machine instruction which removes such continuation from the stack

is POPCONT. So, in order to reach c_{final} an intermediate configuration $c_2 \equiv ([\texttt{POPCONT}],\ h',\ k',\ k',\ v_1 : (k_0, \mathbf{p}) : S),\ cs'[\mathbf{p} \mapsto is_2])$ must be reached so that the execution can proceed. By Lemma 27, $k' = k$.

In fact, it holds that $c_1 \rightarrow^*_{(k_0,\mathbf{p}):S} c_2$, so we can apply i.h. to obtain $E \vdash (h, k), 0, e_1 \Downarrow (h', k), v_1, (\delta_1, m_1, s_1)$, as $(k_0, \mathbf{p}) : S = \mathit{drop}\ 0\ ((k_0, \mathbf{p}) : S)$.

Notice also that $E \uplus [x_1 \mapsto v_1] \equiv \rho + [x_1 \mapsto 1]$ and that $\mathit{drop}\ (td + 1)\ (v_1 : S) = \mathit{drop}\ td\ S = S'$, so we can also apply i.h. to $c_3 \rightarrow^*_{S'} c_{final}$ to obtain $E \uplus [x_1 \mapsto v_1] \vdash (h', k), td+1, e_2 \Downarrow (h'', k), v, (\delta_2, m_2, s_2)$. The reasoning about resources consumption is the same as the opposite implication.

□