Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 246 (2009) 167–182
www.elsevier.com/locate/entcs

A Resource-Aware Semantics and Abstract Machine for a Functional Language with Explicit Deallocation

Manuel Montenegro[2] Ricardo Peña[1,3] Clara Segura[1,3]

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Abstract

Safe is a first-order eager language with heap regions and unusual facilities such as programmer-controlled destruction and copying of data structures. The regions are disjoint parts of the heap where the compiler may allocate data structures. Thanks to regions, a runtime garbage collector is not needed. The language and its associated type system, guaranteeing that destruction facilities and region management are done in a safe way, have been presented previously.
In this paper, we start from a high-level big-step operational semantics for *Safe*, and in a series of semi-formal steps we derive its compilation to an imperative language and imperative abstract machine. Once the memory needs of the machine are known, we enrich the semantics with memory consumption annotations and prove that the enriched semantics is correct with respect to the translation and the abstract machine. All the steps are derived in such a way that it is easy to understand the translation and to formally establish its correctness.

Keywords: Functional languages, Region based heaps, Abstract machines, Code generation.

1 Introduction

Safe is a first-order eager functional language with facilities for programmer-controlled destruction and copying of data structures. It provides also *regions*, i.e. disjoint parts of the heap where the compiler allocates data structures. The allocation and deallocation of such regions are associated with function applications. The *Safe* language and a sharing analysis for it were published in [11]. We also defined a type system and a type inference algorithm [10,9] guaranteeing that destruction facilities and region management are done in a safe way.

[1] Partially supported by the Madrid Region Government under grant S-0505/TIC/0407 (PROMESAS).
[2] Email: montenegro@fdi.ucm.es. Work supported by the MEC FPU grant AP2006-02154.
[3] Email: ricardo@sip.ucm.es, csegura@sip.ucm.es

doi:10.1016/j.entcs.2009.07.021

Apparently, the language is impure as cell and region destruction, if used without rectrictions, is a (very dangerous) side-effect. But if we consider only those programs accepted by such a type system, then the language is pure and side-effects free.

In this paper we derive an imperative machine from a high-level big-step operational semantics and give the function that translates *Safe* programs to imperative code for that machine. The derivation is achieved by incremental refinements across a small-step operational semantics and an intermediate abstract machine.

Once the memory needs of the machine are known, we enrich the semantics with memory consumption annotations and prove that the translation and the abstract machine are correct with respect to the enriched semantics.

We have also implemented a further code generation phase from the last machine presented here (called SVM) to bytecode of the Java Virtual Machine. *Safe* is part of a Proof Carrying Code project and the aim is producing this bytecode together with a formal certificate. In our case, the certificate will prove that the execution of the code is free from dangling pointers.

In Section 2 we give a brief description of the language. Sections 3 and 4 respectively describe a big-step operational semantics and an equivalent small-step operational semantics. Section 5 describes an abstract machine, called SAFE-M2, where a stack of continuations is used. Section 6 presents the imperative machine SVM, and Section 7 the translation schemes from *Safe* to imperative code. A detailed example is given, where efficient tail recursion is apparent. In Section 8 we provide the enriched big-step semantics and a proof that its resource annotations reflect the real consumptions done by the translated program. Finally, in Section 9 we survey some related work and conclude.

2 Summary of *Safe*

Safe is a first-order polymorphic functional language whose syntax is similar to that of (first-order) Haskell or ML, and has some facilities to manage memory. The memory model is based on heap regions where data structures are built. However, in *Full-Safe* in which programs are written, regions are implicit. These are inferred when *Full-Safe* is desugared into *Core-Safe* [8]. As the semantics presented in this paper are defined at *Core-Safe* level, we describe it in detail.

The allocation and deallocation of regions is bound to function calls: a *working region* is allocated when entering the call and deallocated when exiting it. Inside the function, data structures may be built but they can also be destroyed by using a destructive pattern matching denoted by ! or a **case!** expression, which deallocates the cell corresponding to the outermost constructor. Using recursion, the recursive portions of the whole data structure may be deallocated. We say that it is *condemned*. As an example, we show in *Full-Safe* an append function destroying the first list's spine, while keeping its elements in order to build the result:

```
concatD []!     ys  = ys
concatD (x:xs)! ys  = x : concatD xs ys
```

As a consequence, appending needs constant heap space, while the usual version

needs linear heap space. The fact that the first list is lost is reflected in the type of the function: `concatD :: [a]! -> [a] -> [a].`

The data structures which are not part of the function's result are built in the local working region, which we call *self*, and they die when the function terminates. As an example we show a destructive version of the treesort algorithm:

```
treesortD :: [Int]! -> [Int]
treesortD xs = inorder (mkTreeD xs)
```

First, the original list `xs` is used to build a search tree by applying function `mkTreeD` (defined below). This tree is then traversed in inorder to produce the sorted list. The tree is not part of the result of the function, so it will be built in the working region and will die when the `treesortD` function returns (in *Core-Safe* where regions are explicit this will be apparent). The original list is destroyed and the destructive appending function is used in the traversal so that constant heap space is consumed.

Function `mkTreeD` inserts each element of the list in the binary search tree.

```
mkTreeD :: [Int]! -> BSTree Int
mkTreeD []!      = Empty
mkTreeD (x:xs)!  = insertD x (mkTreeD xs)
```

The function `insertD` is the destructive version of insertion in a binary search tree. Then `mkTreeD` exactly consumes the space occupied in the heap by the list. The nondestructive version of this function would consume in the worst case quadratic heap space.

```
insertD :: Int -> BSTree Int! -> BSTree Int
insertD x Empty! = Node Empty x Empty
insertD x (Node lt y rt)!
          | x == y = Node lt! y  rt!
          | x > y  = Node lt! y (insertD x rt)
          | x < y  = Node (insertD x lt) y rt!
```

Notice in the first guard, that the cell just destroyed must be built again. When a data structure is condemned its recursive children may subsequently be destroyed or they may be reused as part of the result of the function. We denote the latter with a !, as shown in this function `insertD`. This is due to safety reasons: a condemned data structure cannot be returned as the result of a function, as it potentially may contain dangling pointers. Reusing turns a condemned data structure into a safe one. The original reference is not accessible any more. So, in the example `lt` and `rt` are condemned and they must be reused in order to be part of the result.

Data structures may also be copied denoted appending @ to a variable. Only the recursive part of the structure is copied, while the elements are shared with the old one. This is useful when we want non-destructive versions of functions based on the destructive ones. For example, we can define `treesort xs = treesortD (xs@).`

In Fig. 1 we show the syntax of *Core-Safe*. A program *prog* is a sequence of possibly recursive polymorphic data and function definitions followed by a main expression e using them, whose value is the program result. The abbreviation $\overline{x_i}^n$ stands for $x_1 \cdots x_n$. Destructive pattern matching is desugared into **case!** expressions. Constructions are only allowed in **let** bindings, and atoms are used in function applications, **case**/**case!** discriminant, copy and reuse. Regions are explicit in constructor application and the copy expression. Function definitions have additional region parameters ${r_j}^l$ where data structures may be built. In the right hand side

$$
\begin{array}{lll}
prog & \to & \overline{data_i}^{\,n}; \overline{dec_j}^{\,m}; e \\
data & \to & \textbf{data}\ T\ \overline{\alpha_i}^{\,n}\ @\ \overline{\rho_j}^{\,m} = \overline{C_k\ \overline{t_{ks}}^{\,n_k}\ @\ \rho_m}^{\,l} \quad \{\text{recursive, polymorphic data type}\} \\
dec & \to & f\ \overline{x_i}^{\,n}\ @\ \overline{r_j}^{\,l} = e \quad \{\text{recursive, polymorphic function}\} \\
e & \to & a \quad \{\text{atom: literal } c \text{ or variable } x\} \\
 & | & x@r \quad \{\text{copy}\} \\
 & | & x! \quad \{\text{reuse}\} \\
 & | & f\ \overline{a_i}^{\,n}\ @\ \overline{r_j}^{\,l} \quad \{\text{function application}\} \\
 & | & \textbf{let}\ x_1 = be\ \textbf{in}\ e \quad \{\text{non-recursive, monomorphic}\} \\
 & | & \textbf{case}\ x\ \textbf{of}\ \overline{alt_i}^{\,n} \quad \{\text{read-only case}\} \\
 & | & \textbf{case!}\ x\ \textbf{of}\ \overline{alt_i}^{\,n} \quad \{\text{destructive case}\} \\
alt & \to & C\ \overline{x_i}^{\,n} \to e \\
be & \to & C\ \overline{a_i}^{\,n}\ @\ r \quad \{\text{constructor application}\} \\
 & | & e
\end{array}
$$

Fig. 1. *Core-Safe* language definition

expression only the r_j and its working region *self* may be used. Functional types include region parameter types.

Polymorphic algebraic data types are defined through **data** declarations. Algebraic types declarations have, after region inference, additional type variables indicating the regions where the constructed values of that type are allocated. Region inference also adds region arguments to constructors, forcing the restriction that recursive substructures must live in the same region as their parent. For example, after region inference, trees are represented as follows:

```
data BSTree a @ rho = Empty@rho | Node (BSTree a@rho) a (BSTree a@rho) @ rho
```

There may be several region parameters when nested types are used: different components of the data structure may live in different regions. In that case the last region variable is the *outermost region* where the constructed values of this type are allocated. In the following example

```
data T a b @ rho1 rho2 = C1 ([a] @ rho1) @ rho2 | C2 b @ rho2
```

`rho2` is where the constructed values of type T are allocated, while `rho1` is where the list of a `C1` value is allocated.

Function `splitD` is an example of function with several output regions. In order to save space we show here a semi-desugared version with explicit regions. Notice that the resulting tuple and its components may live in different regions:

```
splitD :: Int -> [a]!@rho2 -> rho1 -> rho2 -> rho3 -> ([a]@rho1, [a]@rho2)@rho3
splitD 0 zs!     @ r1 r2 r3 = ([]@r1, zs!)@r3
splitD n []!     @ r1 r2 r3 = ([]@r1, []@r2)@r3
splitD n (y:ys)! @ r1 r2 r3 = ((y:ys1)@r1, ys2)@r3
          where (ys1, ys2) = splitD (n-1) ys @r1 r2 r3
```

3 Big-step semantics

In Fig. 2 we show the big-step operational semantics of the core language expressions. We use $v, v_i, \ldots$ to denote values, i.e. either heap pointers or basic constants, and $p, p_i, q, \ldots$ to denote heap pointers. We use $a, a_i, \ldots$ to denote atoms, i.e. either program variables or basic constants. The former are denoted by $x, x_i, \ldots$ and the latter by c, c_i etc. Finally, we use $r, r_i, \ldots$ to denote region variables.

A judgement of the form $E \vdash h, k, e \Downarrow h', k', v$ means that expression e is suc-

$$E \vdash h, k, c \Downarrow h, k, c \quad [Lit]$$

$$E[x \mapsto v] \vdash h, k, x \Downarrow h, k, v \quad [Var_1]$$

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

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

$$\frac{(f\ \overline{x_i}^n @\ \overline{r_j}^m = e) \in \Sigma \quad [x_i \mapsto E(a_i)^n, r_j \mapsto E(r'_j)^m, self \mapsto k+1] \vdash h, k+1, e \Downarrow h', k'+1, v}{E \vdash h, k, f\ \overline{a_i}^n @\ \overline{r'_j}^m \Downarrow h' \mid_{k'}, k', v} \; [App]$$

$$\frac{E \vdash h, k, e_1 \Downarrow h', k', v_1 \quad E \cup [x_1 \mapsto v_1] \vdash h', k', e_2 \Downarrow h'', k'', v}{E \vdash h, k, \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 \Downarrow h'', k'', v} \; [Let_1]$$

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

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

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

Fig. 2. Operational semantics of *Safe* expressions

cessfully reduced to normal form v under runtime environment E and heap h with $k+1$ regions, ranging from 0 to k, and that a final heap h' with $k'+1$ regions is produced as a side effect. Runtime environments E map program variables to values and region variables to actual region identifiers. We adopt the convention that for all E, if c is a constant, $E(c) = c$.

A heap h is a finite mapping from fresh variables p (we call them heap pointers) to construction cells w of the form $(j, C\,\overline{v_i}^n)$, meaning that the cell resides in region j. We say that $region(w) = j$. Actual region identifiers j are just natural numbers. Formal regions appearing in a function body are either region variables r corresponding to formal arguments or the constant *self*. Deviating from other authors, by $h[p \mapsto w]$ we denote a heap h where the binding $[p \mapsto w]$ is highlighted. On the contrary, by $h \uplus [p \mapsto w]$ we denote the disjoint union of heap h with the binding $[p \mapsto w]$. By $h \mid_k$ we denote the heap obtained by deleting from h those bindings living in regions greater than k, and by $dom(h)$, the set $\{p \mid [p \mapsto w] \in h\}$.

The semantics of a program is the semantics of the main expression in an environment Σ, which is the set containing all the function and data declarations.

Rules *Lit* and Var_1 just say that basic values and heap pointers are normal forms. Rule Var_2 executes a copy expression 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 informaton is available in our runtime system so that this function can be implemented. The pointers in non recursive positions are kept identical in the new cells. This implies that both data structures may share some subparts.

In rule Var_3, the binding $[p \mapsto w]$ in the heap is deleted and a fresh binding $[q \mapsto w]$ to cell w is added. This action may create dangling pointers in the live heap, as some cells may contain free occurrences of p.

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. 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 another source of possible dangling pointers.

Rules Let_1, Let_2, and *Case* are the usual ones for an eager language, while rule *Case*! expresses what happens in a destructive pattern matching: the binding of the discriminant variable disappears from the heap. This action is the last source of possible dangling pointers.

In the following, we will feel free to write the derivable judgements as $E \vdash h, k, e \Downarrow h', k, v$ because of the following:

Proposition 3.1 *If $E \vdash h, k, e \Downarrow h', k', v$ is derivable, then $k = k'$.*

Proof. Straightforward, by induction on the depth of the derivation. □

Proposition 3.2 *If e_0 is the main expression of a Safe program, and $[self \mapsto 0] \vdash \{\}, 0, e_0 \Downarrow h_f, 0, v_f$ is derivable, then in every judgement $E \vdash h, k, e \Downarrow h', k, v$ of the derivation $E(self) = k$ holds.*

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

4 Small-Step Semantics

In Figure 3 we show the small-step semantic rules. There are two kinds of judgements. The first kind, $E, h, k_0, k, e \longrightarrow h', k_0, v$, is applied when an expression e is evaluated to a value in one step. These correspond to literals, variables, copy expressions, and reuse expressions. The other kind, $E, h, k_0, k, e \longrightarrow E', h', k_0, k', e'$, covers the remaining cases: function application, **let**, **case** and **case!** expressions. In the configurations, k denotes the highest region available in h, as in the big step semantics. We explain below the meaning of k_0.

Notice that **let** expressions are marked with a natural number δ and an environment E. In rule *App*, the number of available regions is incremented by one, as a new local region is allocated and assigned number $k+1$. Additionally, the environment E is discarded, as in the function body only the arguments and the **self** region are in scope. However, due to **let** expressions, a continuation is possible after function application. Then, we need to recover the discarded environment and the original value of k. The environment is kept in the binding and number δ is used to remember the newly created regions during the evaluation of the bound expression, so that the original k can be later recovered. The initial values of δ and E are respectively 0 and $\perp$, which we can assume are annotated in the text. Rule Let_{4b}

$$\frac{k \geq k_0}{E, h, k_0, k, c \longrightarrow h|_{k_0}, k_0, c}\ [Lit]$$

$$\frac{k \geq k_0}{E[x \mapsto v], h, k_0, k, x \longrightarrow h|_{k_0}, k_0, v}\ [Var_1]$$

$$\frac{k \geq k_0 \quad k \geq j \quad (h', q) = copy(h, p, j)}{E[x \mapsto p, r \mapsto j], h, k_0, k, x@r \longrightarrow h', k_0, q}\ [Var_2]$$

$$\frac{k \geq k_0 \quad fresh(q)}{E[x \mapsto p], h \uplus [p \mapsto w], k_0, k, x! \longrightarrow h \uplus [q \mapsto w], k_0, q}\ [Var_3]$$

$$\frac{(f\ \overline{x_i}^n @\ \overline{r_j}^m = e) \in \Sigma}{E, h, k_0, k, f\ \overline{a_i}^n @\ \overline{r'_j}^m \longrightarrow [x_i \mapsto E(a_i)^n, r_j \mapsto E(r'_j)^m, self \mapsto k+1], h, k_0, k+1, e}\ [App]$$

$$\frac{j \leq k \quad fresh(p)}{E[r \mapsto j, a_i \mapsto {v_i}^n], h, k_0, k, \mathbf{let}\ x_1 =^{\perp}_0 C\ \overline{a_i}^n @ r\ \mathbf{in}\ e \longrightarrow E \cup [x_1 \mapsto p], h \uplus [p \mapsto (j, C\ \overline{v_i}^n)], k_0, k, e}\ [Let_3]$$

$$\frac{E, h, k, k, e_1 \longrightarrow h', k, v_1}{E, h, k_0, k, \mathbf{let}\ x_1 =^{\perp}_0 e_1\ \mathbf{in}\ e \longrightarrow E \cup [x_1 \mapsto v_1], h', k_0, k, e}\ [Let_{4a}]$$

$$\frac{E, h, k, k, e_1 \longrightarrow E', h', k, k+\eta, e'_1}{E, h, k_0, k, \mathbf{let}\ x_1 =^{\perp}_0 e_1\ \mathbf{in}\ e \longrightarrow E', h', k_0, k+\eta, \mathbf{let}\ x_1 =^{E}_{\eta} e'_1\ \mathbf{in}\ e}\ [Let_{4b}]$$

$$\frac{E'' \neq \perp \quad E, h, k, k+\delta, e_1 \longrightarrow h', k, v_1}{E, h, k_0, k+\delta, \mathbf{let}\ x_1 =^{E''}_{\delta} e_1\ \mathbf{in}\ e \longrightarrow E'' \cup [x_1 \mapsto v_1], h', k_0, k, e}\ [Let_{4c}]$$

$$\frac{E'' \neq \perp \quad E, h, k, k+\delta, e_1 \longrightarrow E', h', k, k+\eta, e'_1}{E, h, k_0, k+\delta, \mathbf{let}\ x_1 =^{E''}_{\delta} e_1\ \mathbf{in}\ e \longrightarrow E', h', k_0, k+\eta, \mathbf{let}\ x_1 =^{E''}_{\eta} e'_1\ \mathbf{in}\ e}\ [Let_{4d}]$$

$$\frac{C = C_r}{E[x \mapsto p], h[p \mapsto (j, C\ \overline{b_i}^{n_r})], k_0, k, \mathbf{case}\ x\ \mathbf{of}\ C_i\ \overline{x_{ij}}^{n_i} \to {e_i}^m \longrightarrow E \cup [x_{ri} \mapsto {v_i}^{n_r}], h, k_0, k, e_r}\ [Case]$$

$$\frac{C = C_r}{E \cup [x_{ri} \mapsto {v_i}^{n_r}], h \cup [p \mapsto (j, C\ \overline{b_i}^{n_r})], k_0, k, \mathbf{case!}\ x\ \mathbf{of}\ C_i\ \overline{x_{ij}}^{n_i} \to {e_i}^m \longrightarrow E \cup [x_{ri} \mapsto {v_i}^{n_r}] h, k_0, k, e_r}\ [Case!]$$

Fig. 3. Small-step operational semantics of *Safe* expressions

saves the environment for the first time and rule Let_{4d} updates the information as necessary during the evaluation of the bound expression. In case the evaluation of the bound expression is successful, rules Let_3, Let_{4a} or Let_{4c} will be applied to proceed with the evaluation of the main expression.

Those new regions created during the evaluation of the bound expression cannot contain the result of the evaluation because after function application the local region is deallocated. Region k_0 denotes the highest region available when the machine stops reducing the expression. Initially $k = k_0 = 0$. Rule App increments k while rules Lit, Var_1, Var_2 and Var_3 discard all the local regions back to k_0.

This small-step semantics is equivalent to the previously defined big-step semantics: for any k and $k_0 \leq k$, $\Delta, k, e \Downarrow \Theta, k, v$ if and only if $\Delta, k_0, k, e \longrightarrow^* \Theta, k_0, k, v$.

5 The abstract machine SAFE-M2

Our next refinement is introducing an abstract machine, called SAFE-M2 because there was a previous one called SAFE-M1 now abandoned. A configuration of the machine is a 7-tuple $(h, k_0, k, e, E, S, \Sigma)$, where h is the heap, k_0, k are the region numbers used in the small-step semantics, e is the control expression, E is the runtime environment, S is a stack, and Σ is a function giving the code of every defined *Safe* function. In Figure 4 we show the transitions of the abstract machine

Initial/final configuration	Condition	Label
$(h,\ k_0,\ k,\ c,\ E,\ S,\ \Sigma)$ $\Rightarrow (h\,|_{k_0},\ k_0,\ k_0,\ c,\ E,\ S,\ \Sigma)$	$k > k_0$	$[Lit_1]$
$(h,\ k,\ k,\ c_1,\ E_1,\ (k_0, x_1, e, E) : S,\ \Sigma)$ $\Rightarrow (h,\ k_0,\ k,\ e,\ E \cup [x_1 \mapsto c_1],\ S,\ \Sigma)$		$[Lit_2]$
$(h[p \mapsto (j, C\,\overline{b_i}^n)],\ k_0,\ k,\ x,\ E[x \mapsto p],\ S,\ \Sigma)$ $\Rightarrow (h\,|_{k_0},\ k_0,\ k_0,\ x,\ E,\ S,\ \Sigma)$	$k > k_0$	$[Cons_1]$
$(h[p \mapsto (j, C\,\overline{b_i}^n)],\ k,\ k,\ x,\ E_1[x \mapsto p],\ (k_0, x_1, e, E) : S,\ \Sigma)$ $\Rightarrow (h,\ k_0,\ k,\ e,\ E \cup [x_1 \mapsto p],\ S,\ \Sigma)$		$[Cons_2]$
$(h[p \mapsto (l, C\,\overline{b_i}^n)],\ k_0,\ k,\ x@r,\ E[x \mapsto p, r \mapsto j],\ S,\ \Sigma)$ $\Rightarrow (h',\ k_0,\ k,\ y,\ E \cup [y \mapsto q],\ S,\ \Sigma)$	$(h', q) = copy(h, p, j)$ $j \leq k, fresh(y)$	$[Copy]$
$(h \uplus [p \mapsto w],\ k_0,\ k,\ x!,\ E[x \mapsto p],\ S,\ \Sigma)$ $\Rightarrow (h \uplus [q \mapsto w],\ k_0,\ k,\ y,\ E \cup [y \mapsto q],\ S,\ \Sigma)$	$fresh(q), fresh(y)$	$[Reuse]$
$(h,\ k_0,\ k,\ f\ \overline{a_i}^n\ @\ \overline{s_j}^m,\ E,\ S,\ \Sigma)$ $\Rightarrow (h,\ k_0,\ k+1,\ e,\ [x_i \mapsto E(a_i)^n, r_j \mapsto E(s_j)^m, self \mapsto k+1],\ S,\ \Sigma)$	$(f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e) \in \Sigma$	$[App]$
$(h,\ k_0,\ k,\ \textbf{let}\ x_1 = C\,\overline{a_i}^n @ s\ \textbf{in}\ e,\ E,\ S,\ \Sigma)$ $\Rightarrow (h \uplus [p \mapsto (E(s), C\,E(a_i)^n)],\ k_0,\ k,\ e,\ E \uplus [x_1 \mapsto p],\ S,\ \Sigma)$	$E(s) \leq k$ $fresh(p)$	$[Let_3]$
$(h,\ k_0,\ k,\ \textbf{let}\ x_1 = e_1\ \textbf{in}\ e,\ E,\ S,\ \Sigma)$ $\Rightarrow (h,\ k,\ k,\ e_1,\ E,\ (k_0, x_1, e, E) : S,\ \Sigma)$		$[Let_4]$
$(h[p \mapsto (j, C\,\overline{b_i}^n)],\ k_0,\ k,\ \textbf{case}\ x\ \textbf{of}\ C_i\ \overline{x_{ij}}^{n_i} \to e_i,\ E[x \mapsto p],\ S,\ \Sigma)$ $\Rightarrow (h,\ k_0,\ k,\ e_r,\ E \cup [x_{rj} \mapsto b_j{}^n],\ S,\ \Sigma)$	$C = C_r$	$[Case_1]$
$(h \uplus [p \mapsto (j, C\,\overline{b_i}^n)],\ k_0,\ k,\ \textbf{case!}\ x\ \textbf{of}\ C_i\ \overline{x_{ij}}^{n_i} \to e_i,\ E[x \mapsto p],\ S,\ \Sigma)$ $\Rightarrow (h,\ k_0,\ k,\ e_r,\ E \cup [x_{rj} \mapsto b_j{}^n],\ S,\ \Sigma)$	$C = C_r$	$[Case_2]$

Fig. 4. The abstract machine SAFE-M2

SAFE-M2. The only new element w.r.t. the small-step semantics is the stack S. It consists of *continuation* frames of the form (k_0, x_1, e, E) corresponding to pending expressions e of a **let** whose auxiliary expression e_1 is under evaluation. Region k_0 is where the normal form of e should be returned, x_1 is the **let**-bound variable free in e, and E is the environment in which e should be evaluated. Corresponding to the inductive semantic rules of the Let_4 group, the abstract machine rule Let_4 pushes a continuation to the stack and proceeds with the evaluation of the auxiliary expression e_1. When the normal form of e_1 is reached in rules Lit_1 and $Cons_1$, the continuation is popped and the machine proceeds with the evaluation of the main expression. We use $a, a_i, \ldots$ to denote either program variables or basic constants.

Notice that the current environment is discarded in rules Lit_2 and $Cons_2$ when a normal form is reached and a continuation must be popped from the stack. Also, it is discarded in rule App when a function body is entered and the formal arguments become the only variables in scope. In Section 7 this will have the important consequence that tail recursion is translated so that only a constant stack space is needed. Notice also in rule Let_4 that the current environment is saved in the stack but it is

not discarded from the control. One important aspect of the translation given in Section 7 is that it manages to avoid this implicit duplication of environments.

The current environment is extended with new bindings in rules Let_3, $Case_1$ and $Case_2$ as soon as let-bound or case-bound variables become free variables in scope in the continuation expression. Also, it is extended in rules *Copy* and *Reuse* with a fresh program variable y. This is merely an artifact due to the fact that a fresh data structure must be referenced in the control expression. Finally, in rules Lit_2 and $Cons_2$, the environment E saved in the continuation must be extended with the new binding introduced by **let**.

6 The imperative abstract machine SVM

We first present our imperative machine and then, in Sec. 7, we will explain how to map M2 to it. A configuration of the machine SVM (*Safe Virtual Machine*) consists of the six components $(is,\ h,\ k_0,\ k,\ S,\ cs)$, where is is the current instruction sequence, and cs is the code store where the instruction sequences resulting from the compilation of program fragments are kept. Now, we will use $p, q, \ldots$ to denote code pointers solved by cs, and $b, b_i, \ldots$ to denote heap pointers or any other item stored in the stack (constants, region numbers or continuations). In Figure 5 we show the semantics of SVM instructions in terms of configuration transitions. 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 top element).

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. Notice that b —which will usually be a value— is left in the stack so that it can be accessed by the continuation. Instruction *PUSHCONT* pushes a continuation. It will be used in the translation of a **let**.

Instructions *COPY* and *REUSE* just mimic the corresponding actions *Copy* and *Reuse* of the abstract machine M2. Instruction *CALL* jumps to a new instruction sequence and creates a new region. Instruction *PRIMOP* operates two basic values located in the stack and replaces them by the result of the operation.

Instruction *MATCH* does a vectored jump depending on the constructor of the matched closure. The vector of sequences pointed to by the p_j corresponds to the compilation of a set of **case** alternatives. Instruction *MATCH*! additionally destroys the matched cell.

Instruction *BUILDENV* 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 on the stack; if it is a basic constant c, it is directly pushed on the stack; if it is the identifier *self*, then the current region number k is pushed on the stack. Instruction *BUILDCLS* allocates fresh memory and constructs a heap value. As *BUILDENV*, it receives a list of keys and uses the same conventions. It

Initial/final configuration	Condition
$(DECREGION : is,\ h,\ k_0,\ k,\ S,\ cs)$ $\Rightarrow (is,\ h\vert_{k_0},\ k_0,\ k_0,\ S,\ cs)$	$k \geq k_0$
$([POPCONT],\ h,\ k,\ k,\ b : (k_0, p) : S,\ cs[p \mapsto is])$ $\Rightarrow (is,\ h,\ k_0,\ k,\ b : S,\ cs)$	
$(PUSHCONT\ p : is,\ h,\ k_0,\ k,\ S,\ cs[p \mapsto is'])$ $\Rightarrow (is,\ h,\ k,\ k,\ (k_0, p) : S,\ cs)$	
$(COPY : is,\ h[b \mapsto (l, C\ \overline{v_i}^n)],\ k_0,\ k,\ b : j : S,\ cs)$ $\Rightarrow (is,\ h',\ k_0,\ k,\ b' : S,\ cs)$	$(h', b') = copy(h, b, j)$ $j \leq k$
$(REUSE : is,\ h \uplus [b \mapsto w],\ k_0,\ k,\ b : S,\ cs)$ $\Rightarrow (is,\ h \uplus [b' \mapsto w],\ k_0,\ k,\ b' : S,\ cs)$	$fresh(b')$
$([CALL\ p],\ h,\ k_0,\ k,\ S,\ cs[p \mapsto is])$ $\Rightarrow (is,\ h,\ k_0,\ k+1,\ S,\ cs)$	
$(PRIMOP \oplus : is,\ h,\ k_0,\ k,\ c_1 : c_2 : S,\ cs)$ $\Rightarrow (is,\ h,\ k_0,\ k,\ c : S,\ cs)$	$c = c_1 \oplus c_2$
$([MATCH\ l\ \overline{p_j}^m],\ h[S!l \mapsto (j, C_r^m\ \overline{v_i}^n)],\ k_0,\ k,\ S,\ cs[p_j \mapsto {is_j}^m])$ $\Rightarrow (is_r,\ h,\ k_0,\ k,\ \overline{b_i}^n : S,\ cs)$	
$([MATCH!\ l\ \overline{p_j}^m],\ h \uplus [S!l \mapsto (j, C_r^m\ \overline{v_i}^n)],\ k_0,\ k,\ S,\ cs[p_j \mapsto {is_j}^m])$ $\Rightarrow (is_r,\ h,\ k_0,\ k,\ \overline{b_i}^n : S,\ cs)$	
$(BUILDENV\ \overline{K_i}^n : is,\ h,\ k_0,\ k,\ S,\ cs)$ $\Rightarrow (is,\ h,\ k_0,\ k,\ Item_k(K_i)^n : S,\ cs)$	(1)
$(BUILDCLS\ C_r^m\ \overline{K_i}^n\ K : is,\ h,\ k_0,\ k,\ S,\ cs)$ $\Rightarrow (is,\ h \uplus [b \mapsto (Item_k(K), C_r^m\ Item_k(K_i)^n)],\ k_0,\ k,\ b : S,\ cs)$	$Item_k(K) \leq k,\ fresh(b)$ (1)
$(SLIDE\ m\ n : is,\ h,\ k_0,\ k,\ \overline{b_i}^m : \overline{b'_i}^n : S,\ cs)$ $\Rightarrow (is,\ h,\ k_0,\ k,\ \overline{b_i}^m : S,\ cs)$	

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

Fig. 5. The abstract machine SVM

also receives the constructor C_r^m of the value.

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

7 Translation to imperative code

The main new idea of the translation is to split the runtime environment of the M2 machine into two environments: a compile-time environment ρ mapping program variables to natural numbers, and the actual runtime environment mapping offsets from the top of the stack to actual heap pointers, basic constants or region numbers.

The ρ environment maps a variable to the position in the stack where its runtime value resides. 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 environment occupies the top m positions of the stack and $\rho[x \mapsto 1]$, 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 into the stack. In the M2 rule Let_4, the environment E pushed into the stack is the same as the environment in which the auxiliary expression e_1 is evaluated. The aim is to share the environment instead of duplicating it, and to push only the remaining parameters in the continuation, i.e. the pair (k_0, e) (the variable x_1 will not in fact be needed, but the compilation will ensure that a pointer to its value will be on top of the stack when the continuation is popped). So, the whole environment ρ will consist of a list of smaller environments $[\delta_1, \ldots, \delta_n]$, each one except the first one δ_1, topped with a continuation. Each individual block 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 $n_1 = 0$. We are assuming that a continuation needs two words in the stack and that the remaining items need one word.

The offset with respect to the top of the stack of a variable x defined in the block k, denoted $\rho\ x$, is computed as follows: $\rho\ x \stackrel{\text{def}}{=} \sum_{i=1}^{k} l_i - \delta_k\ x$.

Only the top environment may be extended with new bindings. There are three operations on compile-time environments:

(i) $((\delta, m, 0) : \rho) + \{x_i \mapsto {j_i}^n\} \stackrel{\text{def}}{=} (\delta \cup \{x_i \mapsto m + {j_i}^n, m + n, 0) : \rho$.

(ii) $((\delta, m, 0) : \rho)^{++} \stackrel{\text{def}}{=} (\{\}, 0, 0) : (\delta, m + 2, 2) : \rho$.

(iii) *topDepth* $((\delta, m, 0) : \rho) \stackrel{\text{def}}{=} m$. Undefined otherwise.

The first one extends the top environment with n new bindings, while the second closes the top environment with a 2-indicator and then opens a new one.

Using these conventions, in Figure 6 we show the translation function *trE* taking a *Core-Safe* expression and giving a list of SVM instructions and a code store. There, *NormalForm* ρ is a compilation macro defined as follows:

$$\begin{aligned} \mathit{NormalForm}\ \rho \stackrel{\text{def}}{=}\ & \mathit{SLIDE}\ 1\ (\mathit{topDepth}\ \rho); \\ & \mathit{DECREGION}; \\ & \mathit{POPCONT} \end{aligned}$$

Notice in function applications that the translation of the body is expected to be found in the code store. This is denoted by highlighting address p.

7.1 Efficient tail recursion: an example

We show here a detailed example, a tail recursive version of the factorial function:

$$\begin{aligned} & \mathit{ifact}\ n\ r = \mathbf{case}\ n\ \mathbf{of} \\ & \qquad\qquad 0 \to r \\ & \qquad\qquad \_ \to \mathbf{let}\ r' = r * n\ \mathbf{in}\ (\mathbf{let}\ n' = n - 1\ \mathbf{in}\ \mathit{ifact}\ n'\ r'); \\ & \mathit{ifact}\ 3\ 1 \end{aligned}$$

$$
\begin{array}{lcl}
trE\ c\ \rho & = & BUILDENV\ [c];\\
 & & NormalForm\ \rho\\
trE\ x\ \rho & = & BUILDENV\ [\rho\ x];\\
 & & NormalForm\ \rho\\
trE\ (x@r)\ \rho & = & BUILDENV\ [\rho\ x, \rho\ r];\\
 & & COPY;\\
 & & NormalForm\ \rho\\
trE\ (x!)\ \rho & = & BUILDENV\ [\rho\ x];\\
 & & REUSE;\\
 & & NormalForm\ \rho\\
trE\ (a_1 \oplus a_2)\ \rho & = & BUILDENV\ [\rho\ a_1, \rho\ a_2];\\
 & & PRIMOP;\\
 & & NormalForm\ \rho\\
trE\ (f\ \overline{a_i}^n\ @\ \overline{s_j}^m)\ \rho & = & BUILDENV\ [\overline{\rho\ a_i}^n, \overline{\rho\ s_j}^m];\\
 & & SLIDE\ (n+m)\ (topDepth\ \rho);\\
 & & CALL\ p\\
\multicolumn{3}{l}{\quad \textbf{where}\ \ (f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e) \in \Sigma}\\
\multicolumn{3}{l}{\qquad cs[p\ \mapsto trE\ e\ [(\{\ r_j \mapsto m-j+1^m, x_i \mapsto n-i+m+1^n\}, n+m, 0)]]}\\
trE\ (\textbf{let}\ x_1 = C_l^m\ \overline{a_i}^n @ s\ \textbf{in}\ e)\ \rho = & & BUILDCLS\ C_l^m\ [(\rho\ a_i)^n]\ (\rho\ s);\\
 & & trE\ e\ (\rho + \{x_1 \mapsto 1\})\\
trE\ (\textbf{let}\ x_1 = e_1\ \textbf{in}\ e)\ \rho & = & PUSHCONT\ p; \qquad \&\ \ cs \cup [p \mapsto trE\ e\ (\rho + \{x_1 \mapsto 1\})]\\
 & & trE\ e_1\ \rho^{+\!\!+}\\
trE\ (\textbf{case}\ x\ \textbf{of}\ \overline{alt_i}^n)\ \rho & = & MATCH\ (\rho\ x)\ \overline{p_i}^n\ \ \&\ \ cs \cup [p_i \mapsto trA\ alt_i\ \rho^n]\\
trE\ (\textbf{case!}\ x\ \textbf{of}\ \overline{alt_i}^n)\ \rho & = & MATCH!\ (\rho\ x)\ \overline{p_i}^n\ \ \&\ \ cs \cup [p_i \mapsto trA\ alt_i\ \rho^n]\\
trA\ (C\ \overline{x_i}^n \to e)\ \rho & = & trE\ e\ (\rho + \{x_i \mapsto n-i+1^n\})
\end{array}
$$

Fig. 6. Translation schemes from normalized *Safe* to SVM instructions

In Figure 7 we show both the corresponding imperative code and an outline of executing *ifact* 3 1. We show, from top to bottom and from left to right, the state of the stack after executing some of the instructions (written above the stack).

It is possible to visualize how tail recursion is efficiently done by means of the *SLIDE* 2 4 instruction which discards the previous (already dead) environment. The stack's depth is the same at each recursive call (second, third and fourth columns).

8 Resource-aware semantics

Once the resource consumption of the SVM is known, we enrich the semantics given in Sec. 3 with a resource vector (δ, m, s) obtained as a side effect of evaluating an expression e. The first component is a partial function $\delta : \mathbb{N} \to \mathbb{Z}$ giving for each region k the signed difference between the cells in the final and initial heaps. A positive difference means that new cells have been created in this region. A negative one, means that some cells have been destroyed. By $dom(\delta)$ we denote the subset of $\mathbb{N}$ in which δ is defined. By $|\delta|$ we mean the sum $\sum_{n \in dom(\delta)} \delta(n)$ giving the total balance of cells. The remaining components m and s respectively give the *minimum* number of fresh cells in the heap and of words in the stack needed to successfully evaluate e. When e is the main expression, these figures give us the total memory needs of the *Safe* program. In Fig. 8, we show the enriched rules. Notice the additional argument *td* needed to simulate the *topDepth* function of compile

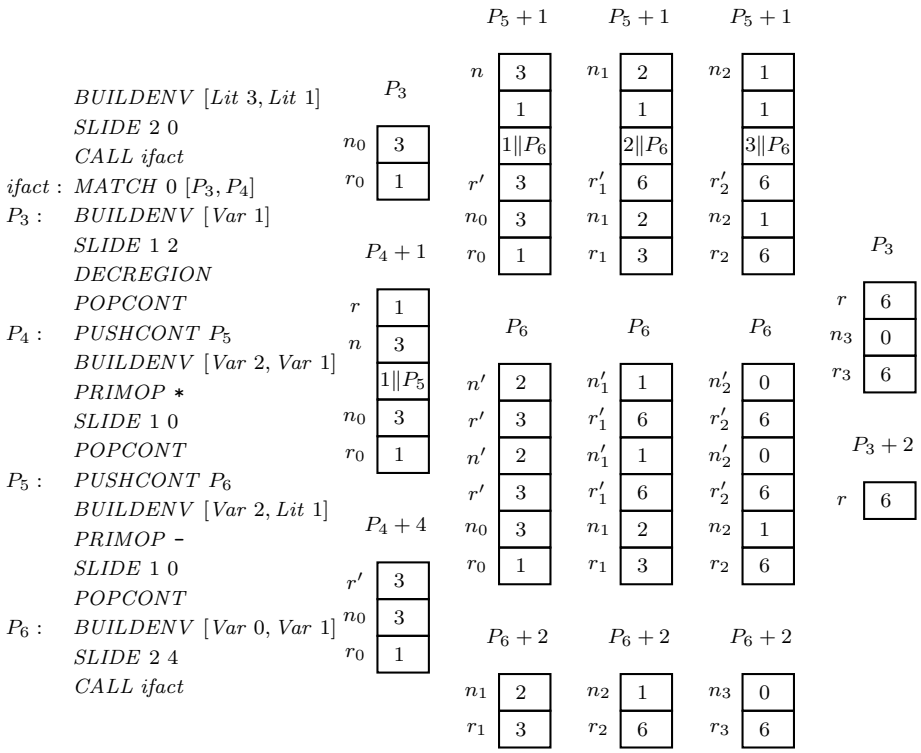

Fig. 7. Imperative code for *ifact* 3 1 and example of execution

time environments. By $[\,]$ we denote the function $\lambda n.\bot$ and by $\delta_1 + \delta_2$ the function:

$$(\delta_1 + \delta_2)(x) = \begin{cases} \delta_1(x) + \delta_2(x) & \text{if } x \in dom(\delta_1) \cap dom(\delta_2) \\ \delta_i(x) & \text{if } x \in dom(\delta_i) - dom(\delta_{3-i}),\ i \in \{1,2\} \\ \bot & \text{otherwise} \end{cases}$$

Function *size* in rule Var_2 gives the size of the recursive spine of a data structure:

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

where *RecPos* returns the recursive parameter positions of a given constructor. In rule *App*, by $\delta|_k$ we mean a function like δ but undefined for values greater than 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. In rule Let_1, a continuation (2 words) is stacked before evaluating e_1, and this a leaves a value in the stack before evaluating e_2. Hence, the computation $\max\{2 + s_1, 1 + s_2\}$.

Now we show that the pair translation-abstract machine is sound and complete with respect this semantics. 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).

Definition 8.1 We say that the environment E and the pair (ρ, S) are equivalent, denoted $E \equiv (\rho, S)$, if $dom\ E - \{self\} = dom\ \rho$, and $\forall x \in dom\ \rho\ .\ E(x) = S!(\rho\ x)$.

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

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

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

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

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

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

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

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

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

Fig. 8. Resource-Aware Operational semantics of *Safe* expressions

Definition 8.2 Given $c_0 = (is, h, k_0, k, S, cs)$ and S' a suffix of S, we denote by $c_0 \rightarrow^*_{S'} c_n$ a derivation in which all the stacks in configurations c_i are never smaller than S'. Should the top instruction of a configuration create a smaller stack, then the machine would stop at that configuration.

Definition 8.3 Given $c_0 = (is, h, k_0, k, S, cs)$ and $c_0 \rightarrow_{S'} \cdots \rightarrow_{S'} c_n$ we call the highest difference in cells between the heaps of the configurations $c_0, \ldots c_n$ and the heap h the *maximum number of fresh cells* of the derivation, denoted $maxFreshCells(c_0 \rightarrow^*_{S'} c_n)$. Likewise, we could 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.

Theorem 8.4 *For all $S, S', E, h, h', td, k_0, k, e, v, \delta, m, s, \rho, cs, cs', cs''$ of their respective types, if*

$$\begin{array}{lll} E \equiv (\rho, S) & (is, cs) = trE\ e\ \rho & td = topDepth\ \rho \\ S' = drop\ td\ S & cs'' = cs \uplus cs' & k_0 \leq k \end{array}$$

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

$$c_0 \equiv (is, h, k_0, k, S, cs'') \rightarrow^*_{S'} ([POPCONT], h'\ |_{k_0}, k_0, k_0, v : S', cs'') \equiv c_n\ \wedge$$
$$\delta = diff\ k\ h\ h'\ \wedge\ m = maxFreshCells(c_0 \rightarrow^*_{S'} c_n)\ \wedge\ s = maxFreshWords(c_0 \rightarrow^*_{S'} c_n)$$

Proof. By induction on the depth of the $\Downarrow$ derivation the ($\Rightarrow$) direction, and by induction on the number of steps of $\rightarrow^*_{S'}$, the ($\Leftarrow$) direction (see [7] for a full proof).□

9 Conclusions and related work

The motivation for this work has been to complete the implementation of *Safe*, whose front-end has been presented in [11,10,9]. One contribution is, in our view, to show 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. Another one, is presenting a semantics enriched with memory costs and proving the correctness of these costs and of the whole translation of *Safe* to imperative code. This semantics will be the basis for proving correct a memory consumption static analysis which we are completing.

There have been other successful derivations of abstract machines starting from high level descriptions of the semantics. For instance, in [4] and [1] 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 [5] 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.

For the first abstract machine M2 we have found inspiration in Sestoft's derivation of abstract machines for a lazy λ-calculus [12]. For the rest of the derivation, the authors have reported some previous experience in [3], 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. For the semantics enriched with a resource vector, we have found inspiration in [2].

Compared to other eager machines such as Landin's SECD machine [6], 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 the example of Section 7.1. 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.

Additionally, our SVM machine does not need a garbage collector and all memory allocation/deallocation actions have been implemented in constant time.

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A Functional Correspondence between Evaluators and Abstract Machines. In *Proceedings of PPDP'03*, pages 8–19. ACM Press, 2003.

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

[3] A. de la Encina and R. Peña. From Natural Semantics to C: A Formal Derivarion of two STG Machines. *Journal of Functional Programming, Cambridge University Press*, pages 1–48, 2008. (to appear).

[4] J. Hannan and D. Miller. From Operational Semantics to Abstract Machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

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

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

[7] M. Montenegro, R. Peña, and C. Segura. A Resource-Aware Semantics and Abstract Machine for a Functional Language with Explicit Deallocation. Technical report, SIC-7-08, Dpto. de Sistemas Informáticos y Computación. Universidad Complutense de Madrid, 2008. Available at http://federwin.sip.ucm.es/sic/investigacion/publicaciones/informes-tecnicos/.

[8] M. Montenegro, R. Peña, and C. Segura. A Simple Region Inference Algorithm for a First-Order Functional Language. In *9th Symposium on Trends in Functional Programming, TFP'08*, pages 194–208, 2008.

[9] M. Montenegro, R. Peña, and C. Segura. A Type System for Safe Memory Management and its Proof of Correctness. In *10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, PPDP'08*, pages 152–162. ACM SIGPLAN, 2008.

[10] M. Montenegro, R. Peña, and C. Segura. An Inference Algorithm for Guaranteeing Safe Destruction. In *Pre-Proceedings of 18th International Symposum on Logic-Based Program Synthesis and Transformation, LOPSTR'08*, pages 13–27, 2008.

[11] R. Peña, C. Segura, and M. Montenegro. A Sharing Analysis for SAFE. In *Selected Papers of the 7th Symposium on Trends in Functional Programming, TFP'06*, pages 109–128. Intellect, 2007.

[12] P. Sestoft. Deriving a Lazy Abstract Machine. *J. of Functional Programming*, 7(3):231–264, 1997.