VIII Jornadas sobre Programación y Lenguajes
Almendros-Jiménez y Suárez-Cabal Eds.
Gijón, del 8 al 10 de Octubre de 2008

A Certified Implementation of a Functional Virtual Machine on top of the Java Virtual Machine

Ricardo Peña[1,2] Delfín Rupérez[3]

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Abstract

As part of a bigger project, we have defined the virtual machine SVM (*Safe Virtual Machine*) which is the target machine of the Safe compiler. Safe is a first-order functional language with unusual memory management features: memory is explicitly and implicitly deallocated at some specific points in the program text, and there is no need for a runtime garbage collector.
We have implemented the SVM on top of the Java Virtual Machine (JVM) because Safe is embedded in a *Proof Carrying Code (PCC)* framework and the final code emitted by the Safe compiler should be Java bytecode. As one of the aims of Safe is providing certificates (mathematical proofs) for some program properties, we require this implementation to be also certified, i.e. the semantics of Safe must be preserved across all the translations. We have used the proof assistant Isabelle/HOL for this purpose.
The paper presents ongoing work on this implementation: first an introduction to Safe and SVM is done; then the main decisions in order to map the SVM data structures to the JVM are explained. Finally, the Isabelle/HOL formalisations of SVM, JVM, and of the mapping between both are presented, and the main correctness theorem is stated.

Keywords: Virtual machines, Java bytecode, proof carrying code, certification.

1 Introduction

Safe is a first-order eager language with facilities for programmer controlled destruction and copying of data structures [8]. Its runtime system uses *regions*, i.e. disjoint parts of the heap where the program allocates data structures, and it does not need a garbage collector. The language is aimed at inferring and certifying upper bounds for memory consumption in a Proof Carrying Code environment [9]. Some of its analyses have been presented elsewhere [12,6].

The smallest memory unit is the *cell*, a contiguous memory space big enough to hold a data construction. A cell contains the mark of the constructor and a representation of the free variables to which the constructor is applied. These may

[1] Partially supported by the Madrid Region Government under grant S-0505/TIC/0407 (PROMESAS).
[2] Email: `ricardo@sip.ucm.es`
[3] Email: `de300379@hotmail.com`

consist, either of basic values, or of pointers to other constructions. It is allocated at constructor application time. A *region* is a collection of cells. It is created empty and it may grow and shrink while it is active. Region deallocation frees all its cells. The allocation and deallocation of regions is bound to function calls. A *working region*, denoted by the reserved identifier *self*, is allocated when entering the call and deallocated when exiting it. Inside the function, data structures not belonging to the output may be built there. When a function body is executing, the *live* regions are the working regions of all the active calls leading to this one. Not all live regions are in scope: they are (for reading) those regions where the arguments live, and (for reading and writing) the regions received as additional arguments, and the current *self* region. The region arguments are explicit in the intermediate code but not in the source: they are inferred by the compiler. The following list sorting function builds and intermediate tree not needed in the output:

```
treesort xs = inorder (makeTree xs)
```

After region inference, the code is annotated with region arguments (after the `@`):

```
treesort xs @ r = inorder (makeTree xs @ self) @ r
```

so that the tree is created in `treeSort` *self* region and deallocated upon termination.

Destruction facilities are also available in the language. For instance, we show here a constant space funcion appending two lists:

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

The ! mark is the way programmers indicate that the left list must be destroyed. This feature and the type system allowing to use it in a safe way have been explained in previous papers [8,6]. The constant space consumption is due to that, at each recursive call, a cell is deleted by the pattern matching while a new one is allocated by the (:) construction.

Isabelle/HOL [10] is a well-known proof assistant, allowing to express definitions and properties in a formal language and to prove them with some human help. It provides lots of proving methods among which the user may choose at each proof step. We have formalised in Isabelle/HOL the semantics of our intermediate language Safe-Imp and of its corresponding abstract machine SVM (*Safe Virtual Machine*), its translation to the Java Virtual Machine (JVM), and the JVM itself by extending a previous formalisation by G. Klein [2]. This allows us to formally state and prove the correctness of our implementation.

2 The Safe Virtual Machine

The Safe compiler translates the Haskell-like source language shown in Section 1 into a set of sequences of imperative Safe-Imp instructions. These belong to the instruction set, presented in Figure 1, of the SVM. A configuration of the SVM consists of the six components $(is,\ \Delta,\ k_0,\ k,\ S,\ cs)$, where is is the current instruction sequence, Δ is the heap, k_0 and k are machine registers denoting regions, S is the stack and cs is the code store where the instruction sequences are kept.

A heap Δ is a function from pointers to construction cells w of the form $(j, C\,\overline{b_i}^{n})$, meaning that the cell is located in region j, that C is the data constructor and the

Initial/final configuration	**Condition**
$(\texttt{DECREGION} : is,\ \Delta,\ k_0,\ k,\ S,\ cs)$	$k \geq k_0$
$\Rightarrow (is,\ \Delta\vert_{k_0},\ k_0,\ k_0,\ S,\ cs)$	
$([\texttt{POPCONT}],\ \Delta,\ k,\ k,\ b : (k_0, p) : S,\ cs[p \mapsto is])$	
$\Rightarrow (is,\ \Delta,\ k_0,\ k,\ b : S,\ cs)$	
$(\texttt{PUSHCONT}\ p : is,\ \Delta,\ k_0,\ k,\ S,\ cs[p \mapsto is'])$	
$\Rightarrow (is,\ \Delta,\ k,\ k,\ (k_0, p) : S,\ cs)$	
$(\texttt{COPY} : is,\ \Delta[b \mapsto (l, C\,\overline{b_i}^n)],\ k_0,\ k,\ b : j : S,\ cs)$	$(\Theta, b') = \mathit{copy}(\Delta, j, b)$
$\Rightarrow (is,\ \Theta,\ k_0,\ k,\ b' : S,\ cs)$	$l \neq j,\ j \leq k$
$(\texttt{REUSE} : is,\ \Delta \cup [b \mapsto w],\ k_0,\ k,\ b : S,\ cs)$	$\mathit{fresh}(b')$
$\Rightarrow (is,\ \Delta \cup [b' \mapsto w],\ k_0,\ k,\ b' : S,\ cs)$	
$([\texttt{CALL}\ p],\ \Delta,\ k_0,\ k,\ S,\ cs[p \mapsto is])$	
$\Rightarrow (is,\ \Delta,\ k_0,\ k+1,\ S,\ cs)$	
$(\texttt{PRIMOP} \oplus : is,\ \Delta,\ k_0,\ k,\ c_1 : c_2 : S,\ cs)$	$c = c_1 \oplus c_2$
$\Rightarrow (is,\ \Delta,\ k_0,\ k,\ c : S,\ cs)$	
$([\texttt{MATCH}\ l\ \overline{p_j}^m],\ \Delta[S!l \mapsto (j, C_r^m\ \overline{b_i}^n)],\ k_0,\ k,\ S,\ cs[\overline{p_j \mapsto is_j}^m])$	
$\Rightarrow (is_r,\ \Delta,\ k_0,\ k,\ \overline{b_i}^n : S,\ cs)$	
$([\texttt{MATCH!}\ l\ \overline{p_j}^m],\ \Delta \cup [S!l \mapsto (j, C_r^m\ \overline{b_i}^n)],\ k_0,\ k,\ S,\ cs[\overline{p_j \mapsto is_j}^m])$	
$\Rightarrow (is_r,\ \Delta,\ k_0,\ k,\ \overline{b_i}^n : S,\ cs)$	
$([\texttt{MATCHN}\ l\ v\ m\ \overline{p_j}^m],\ \Delta,\ k_0,\ k,\ S,\ cs[\overline{p_j \mapsto is_j}^{m+1}])$	$r = S!l - v + 1 \land 1 \leq r \leq m$
$\Rightarrow (is_r,\ \Delta,\ k_0,\ k,\ S,\ cs)$	
$([\texttt{MATCHN}\ l\ v\ m\ p],\ \Delta,\ k_0,\ k,\ S,\ cs[p \mapsto \overline{is_i}^{m+1}])$	$r = S!l - v + 1 \land \neg(1 \leq r \leq m)$
$\Rightarrow (is_{m+1},\ \Delta,\ k_0,\ k,\ S,\ cs)$	
$(\texttt{BUILDENV}\ \overline{K_i}^n : is,\ \Delta,\ k_0,\ k,\ S,\ cs)$	
$\Rightarrow (is,\ \Delta,\ k_0,\ k,\ \overline{\mathit{Item}_k(K_i)}^n : S,\ cs)$	(1)
$(\texttt{BUILDCLS}\ C_r^m\ \overline{K_i}^n\ K : is,\ \Delta,\ k_0,\ k,\ S,\ cs)$	$\mathit{Item}_k(K) \leq k,\ \mathit{fresh}(b)$
$\Rightarrow (is,\ \Delta \cup [b \mapsto (\mathit{Item}_k(K), C_r^m\ \overline{\mathit{Item}_k(K_i)}^n)],\ k_0,\ k,\ b : S,\ cs)$	(1)
$(\texttt{SLIDE}\ m\ n : is,\ \Delta,\ k_0,\ k,\ \overline{b_i}^m : \overline{b'_i}^n : S,\ cs)$	
$\Rightarrow (is,\ \Delta,\ k_0,\ k,\ \overline{b_i}^m : S,\ cs)$	

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

Fig. 1. The abstract machine SVM

b_i are its arguments. Region identifiers j are natural numbers. By $\Delta\ |_k$ we denote the heap obtained by deleting from Δ those cells living in regions bigger than k. We will use $p, q, \ldots$ to denote code labels solved by cs, and $b, b_i, \ldots$ to denote either cell pointers solved by Δ, or basic constants. In Figure 1 we show the semantics of the SVM instructions in terms of configuration transitions. By C_r^m we denote the data constructor which leads to the r-th alternative out of m of a **case**. 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).

We do not show the translation from Safe to Safe-Imp here but, in order to understand the machine behaviour, we give some hints about it:

- **where** clauses are transformed into **let** expressions and, **let** $x_1 = e_1$ **in** e_2 is translated into a PUSHCONT instruction, pushing a continuation for e_2 in the SVM stack, followed by the instruction sequence for e_1.
- Pattern matching is transformed into **case** expressions and these are translated into a switch instruction of the MATCH group which jumps to the appropriate alternative. Each one is a separated instruction sequence.
- The translation of function application consists of first preparing the appropriate environment in the SVM stack (BUILDENV instruction), and then jumping to the function body (CALL instruction). In SVM, function calls are always tail recursive, so there is no need for a RETURN instruction.
- When a normal form is reached, the following events happen in the machine: (1) the current environment is deleted from the stack (SLIDE instruction); (2) some regions may be deallocated, since a tail recursive call chain terminates (DECREGION instruction); and (3) a continuation sequence is looked for in the stack (POPCONT instruction).

We now explain more closely each individual instruction.

Instruction DECREGION deletes from the heap all the regions, if any, between the current topmost region k and region k_0, excluding the latter.

Instruction POPCONT pops a continuation from the stack or stops the execution if there is none. Notice that b is left in the stack as it may be accessed by the continuation. Instruction PUSHCONT pushes a continuation represented by a region number k_0 and a code label p.

Instructions COPY and REUSE just mimic the corresponding actions of the Safe language. 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 primitive operation.

Instruction MATCH does a vectored jump depending on the constructor of the matched cell. Instruction MATCH! is similar but it additionally destroys the matched cell. It is used when the pattern matching is destructive. Instruction MATCHN is used for **case** expressions with an integer discriminant.

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, the item $S!k_i$ 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 a fresh cell in the heap and fills it with a data construction. As BUILDENV, it receives a list of keys and uses the same conventions. Finally, instruction SLIDE removes some parts of the stack.

The following invariant is ensured by the Safe compiler: *For every instruction sequence in the code store cs, instruction i is the last one if and only if it belongs to the set* {POPCONT, CALL, MATCH, MATCH!, MATCHN}.

We have formalised the SVM in Isabelle/HOL by first defining some datatypes for normal form values, and cells:

datatype $Val = Loc\ Location \mid IntT\ int \mid BoolT\ bool$
types $Cell = Constructor \times Val\ list$

A location is just an Isabelle *nat*. Names for constructors, variables, functions, etc, belong to the Isabelle type *string*. The heap is modelled by a pair consisting of a partial function from locations to pairs $(Region, Cell)$, and a *nat* with the total number of regions. The stack is modelled by a list.

types $HeapMap = Location \rightharpoonup (Region \times Cell)$
$Heap = HeapMap \times nat$
$Stack = StackObject\ list$

where stack objects can be values, region numbers or continuations. The SVM code is modelled by a list of triples, each one consisting of a code label (of type *nat*), a SVM instruction sequence, and a function name. The aim is to represent a partial function, but one which can be traversed. A code store provides also information about which labels correspond to continuations.

types $CodeSequence = SafeInstr\ list$
$SVMCode = (CodeLabel \times CodeSequence \times FunName)\ list$
$ContinuationMap = FunName \rightharpoonup CodeLabel\ list$
$CodeStore = SVMCode \times ContinuationMap$

The program counter of the SVM is a pair $(CodeLabel, nat)$ indicating the instruction sequence under execution and the next instruction of the sequence to be executed. The SVM state consists of the heap (including the number k of regions), the register k_0, the program counter and the stack. Finally, the static part of a SVM program consist of a code store, a constructor table, and a sizes table. The second one is needed in order to get at runtime some constructor static attributes, and the last one consists of maximum sizes computed by the compiler for the heap and the stack.

types $PC = CodeLabel \times nat$
$SVMState = Heap \times Region \times PC \times Stack$
$SafeImpProg = CodeStore \times ConstructorTableType \times SizesTable$

A function $incrPC\ (l, i) = (l, i+1)$ is defined in order to increment the program counter.

We have defined a generic function *execSVM* making the SVM to execute the next instruction, or to stop if there is none:

$execSVM :: SafeImpProg \times SVMState \Rightarrow SVMState\ option$
$execSVM\ (((code, cm), ct, st), (h, k0, (l, i), S)) = execSVMInst\ (the\ (code\ l)\ !\ i)\ ct\ h\ k0\ (l, i)\ S$

There is an equation defining *execSVMInst* for every SVM instruction. The definitions closely follow the semantics given in Figure 1. As an example, we show the equations corresponding to `PUSHCONT` and `POPCONT`:

```
execSVMInst :: [SafeInstr, ConstructorTableType, Heap, Region, PC, Stack] ⇒ SVMState option
execSVMInst (PUSHCONT l) ct h k0 pc S = Some (h, snd h, incrPC pc, Cont (k0, l)#S)
execSVMInst POPCONT ct h k0 pc S = (case S of
                                      v#[] ⇒ None
                                    | v#Cont (k0, l)#S' ⇒ Some (h, k0, (l, 0), v#S'))
```

3 Implementation of the SVM on top of the JVM

The JVM provides support for allocating new objects in the heap but not for releasing them. Instead, there is an automatic garbage collector system which, at unpredictable times, collects unused objects. On the contrary, the SVM has ex-

void PushRegion ()	-- creates a top empty region
void PopRegion ()	-- removes the topmost region
void Decregion ()	-- removes the $k - k_0$ topmost regions
cell ReserveCell ()	-- returns a fresh cell
void InsertCell (p, j)	-- inserts cell p into region j
void ReleaseCell (p)	-- releases cell p
cell Copy (p, j)	-- copies the data structure beginning at p into region j

Fig. 2. The interface of the classes `Heap` and `CellFactory`.

plicit releasing of cells and no garbage collector. The JVM provides a frames stack for invoking and returning from methods. The SVM control flow does not follow the typical call/return scheme of imperative languages. The SVM is a kind of 'jumping machine' where the control flow is in part driven by the stack. So, the JVM stack is not appropriate to be used as the SVM stack. Finally, the SVM stores *code addresses* in the stack which are to be used to jump to the corresponding code. There is no support instruction in the JVM for this need. Summarising, careful design decisions are needed in order to correctly map the SVM to the JVM. First we explain how data structures are mapped, and then how the code is mapped.

3.1 Mapping the SVM data structures

As explained in Section 1, one aim of Safe is to statically infer and certify upper bounds for heap and stack sizes. In order to make reusing released cells easier, we have decided to have fixed-size cells in the heap. The size is determined at compile time for each particular program, according to the biggest size data constructor.

The heap is implemented by two classes, `CellFactory` and `Heap` providing a pool of free cells and a stack of regions, each one consisting of a collection of cells. Before refining this description, let us look at the main interface methods, which we present in Figure 2. Following their order of occurrence, they respectively give support to the SVM instructions `CALL`, `DECREGION` (2 methods), `BUILDCLS` (2 methods), `MATCH!` and `COPY`.

Notice that access to an arbitrary region is needed in *InsertCell* and *Copy*, while *ReleaseCell* is provided with only the cell pointer as an argument. We have implemented all the methods (except *Decregion*) running in constant time by representing the regions and the pool as circular doubly-chained lists. Method *Decregion* has a cost in $\Theta(k - k_0)$ independently of the number of cells of the deleted regions. The region stack is represented by a static array of dynamic lists, so that constant time access to each region is provided. Figure 3 shows a picture of the heap.

Initially, all the cells and the region array are allocated with sizes provided by the compiler. During program execution, 'allocating' and 'releasing' cells mean to move them from/to the freelist to/from the appropriate region list.

The SVM stack is implemented by a class `Stack` having a static array with a size provided by the compiler. Its only meaningful method is *Slide* (m, n), which gives support to the `SLIDE` instruction. The rest of accesses are done by the in-line code emitted by the compiler. We will call RTS (run-time system) to the package

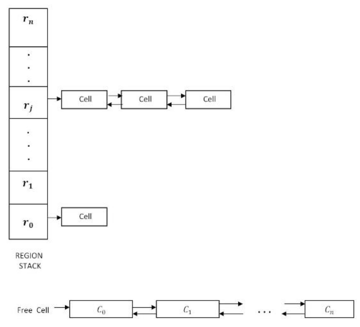

Fig. 3. A picture of the SVM heap and freelist as implemented in the JVM.

consisting of the classes `Heap, CellFactory`, and `Stack`.

3.2 Mapping the SVM code

The code generated by translating the SVM code of a complete Safe program consists of a single JVM class `PSafe` with a single method `PSafeMain()`. Safe functions and the main expression correspond to certain fragments of this method. This decision is forced by the previous one of not using the frames stack of the JVM. Since arguments are pushed to the SVM stack, function calls are implemented by JVM `goto` instructions. A sequence of SVM instructions is represented by a jump-free bytecode sequence. Invocation of RTS methods are allowed in the sequence.

The SVM `MATCH` instructions are implemented by using JVM `Tableswitch` instructions, which can branch in constant time to any label of a list of static labels. The problem of storing/retrieving code addresses into/from the stack is solved in this way: every continuation label p of every Safe function is given a number $i = cm(p)$ in the range $0, \ldots, totC - 1$, being $totC$ the total number of continuations in the program, which in turn is equal to the number of its **let** expressions (so, $totC$ is a static quantity), and being cm an appropriate bijective function. Instruction `PUSHCONT` p just pushes i to the stack, while `POPCONT` uses a global `Tableswitch` instruction indexed by i to jump to the appropriate static label.

4 Formalisation of the JVM in Isabelle/HOL

In the past years, there have been some efforts to formally define the JVM to proof assistants in order to verify properties of the machine itself or of applications written in Java bytecode. Concerning Isabelle/HOL, there was some early work by Cornelia Pusch [13] followed by Tobias Nipkow, Gerwin Klein and others in the framework of some EU-funded projects [11,4,1,14,3]. As starting point, we have used the definition of the JVM done in 2003 by G. Klein for Microjava, a subset of Java [2], and have extended it with a static heap and some instructions such as `Tableswitch`, `Invoke_static`, and others.

A JVM program is formalised as a list of class declarations, each one consisting of the class and superclass names, and two lists for field and method declarations.

```
datatype instr =  Load nat                                    | Jsr int      | Ret nat
                | Store nat                                   | Return       | ArrLoad
                | LitPush val                                 | Pop          | ArrStore
                | New cname                                   | Dup          | ArrLength
                | Getfield vname cname                        | Dup_x1       | ArrNew ty
                | Putfield vname cname                        | Dup_x2       | Checkcast cname
                | Getstatic vname cname                       | Swap         | Tableswitch int int (int list)
                | Putstatic vname cname                       | BinOp op
                | Invoke cname mname (ty list)                | Ifcmpeq int
                | Invoke_static cname mname (ty list)         | Throw
                | Invoke_special cname mname (ty list)        | Goto int
```

Fig. 4. Supported instructions of the JVM.

```
types fdecl = vname × ty                                -- field declaration
      sig = mname × ty list                             -- signature of a method
      'c mdecl = sig × ty × 'c                          -- method declaration ('c is the body's type)
      'c class = cname × fdecl list × 'c mdecl list     -- class = superclass, fields, method
      'c cdecl = cname × 'c class                       -- class declaration
      'c prog = 'c cdecl list                           -- program
```

A method's body provides the lengths of the operand stack and of the local variable list (two Isabelle *nat*), the bytecode instructions, and an exception table.

```
types bytecode = instr list
      jvm_method = nat × nat × bytecode × exception_table
      jvm_prog = jvm_method prog
```

The supported instructions are listed in Figure 4. They represent both a subset and an abstraction of the actual JVM instruction set [5].

Now, we pay attention to the dynamic state of the JVM. It is formed by four components: a possibly raised exception, a static heap, a dynamic heap, and a frames stack. The first contains the address of the exception object, the second is a partial function from pairs ⟨class name, field name⟩ to values, and the third is a partial function from locations to either objects or arrays. An object contains its class name and a mapping from pairs ⟨field name, class name⟩ to values. A frame is formalised as a tuple containing the operand stack, the local variables and arguments, the class name, the method signature, and the program counter.

```
datatype heap_entry = Obj cname (vname × cname ⇀ val) -- class instance with class name and fields
                    | Arr ty nat (nat ⇀ val)           -- array with element type, length, and entries

types sheap = cname × vname ⇀ val                      -- static heap
      dheap = loc ⇀ heap_entry                          -- dynamic heap
      frame = opstack × locvars × cname × sig × pc
      jvm_state = val option × sheap × dheap × frame list
```

Klein defines a function $exec :: jvm\_prog \Rightarrow jvm\_state \Rightarrow jvm\_state\ option$, executing the next instruction in the machine, which we have extended to the added instructions. In addition, we define:

```
execUntil :: jvm_prog ⇒ pc ⇒ jvm_state ⇒ jvm_state
execUntil P pc s = if pc = getPC s then s else execUntil P pc ((the ∘ exec P) s)
```

which executes a JVM program from a given state up to the state (if any) in the current method having *pc* as its program counter.

5 Certification of the Implementation

First, we must provide Isabelle with the translation from SVM code to JVM. In order not to duplicate job and, more importantly, in order to ensure that the actual translation done by the Safe compiler is exactly the one formalised in Isabelle, we have defined this translation in Isabelle and used the code generation facilities of this tool to generate the Haskell code actually executed by the compiler. The translation provides, as add-ons, a function mapping the SVM program labels to the JVM bytecode labels corresponding to the translation, and a function mapping continuation labels to the small integers mentioned in Section 3.2.

$trSVM2JVM :: SafeImpProg \Rightarrow jvm\_prog \times codeMap \times contMap$
$codeMap :: PC \rightharpoonup pc$
$contMap :: CodeLabel \rightharpoonup nat$

Then, we define an equivalence relation between SVM and JVM states, taking into account that part of the SVM state is implemented by the static data structures kept in the RTS classes `Heap` and `Stack`. The relation may consider a SVM state to be equivalent to several JVM states, since for instance there is an abstraction when going from lists of cells in the JVM to set of cells in the SVM. But also, a state in JVM can be considered equivalent to several SVM states: those resulting from applying any bijection to the set of defined heap locations.

Definition 5.1 We say that the SVM state $s = (H, K_0, PC, S)$ and the JVM state $s' = (None, h_s, h_d, ([\,], vs, \text{“PSafe”}, (\text{“PSafeMain”}, ts), pc)\#[\,])$ are equivalent under the SVM program P, denoted $P \vdash s \simeq s'$ if, given $(P', cdMap, ctMap) = trSVM2JVM\ P$, there exists a bijection $g :: Location \Rightarrow loc$ such that:

(i) The heaps H and $(P'.Heap, h_s, h_d)$ are equivalent under g.

(ii) The stacks S and $(P'.Stack, h_s, h_d)$ are equivalent under g and $ctMap$.

(iii) $K_0 = h_s\ (Heap, K_0)$.

(iv) $pc = cdMap\ PC$.

The definitions of the equivalence relations between heaps and between stacks are rather involved. In the heap case, it uses the constructor table of P and expresses that the SVM cell locations mapped by g to JVM heap locations point to equivalent cells, i.e. they contain the same values and live in the same region. In the stack case, it expresses that, position by position, S and its JVM implementation contain either the same two values, or two heap locations made equivalent by g, or two continuations made equivalent by *ctMap*. They are omitted for lack of space.

The main correctnes theorem states that, if the SVM and its implementation are started in equivalent states, then after the SVM executes its next instruction, and after the number of steps required by the JVM to execute its translation, both machines arrive to equivalent states. Formally:

Theorem 5.2 *If $P \vdash s_1 \simeq s'_1$, $execSVM\ (P, s_1) = Some\ s_2$, PC is the program counter of s_2, and $(P', cdMap, ctMap) = trSVM2JVM\ P$, then there exists s'_2 such that:*

(i) $execUntil\ P'\ (cdMap\ PC)\ s'_1 = s'_2$.

(ii) $P \vdash s_2 \simeq s_2'$.

6 Conclusions

We have presented a summary of the formalisations in Isabelle/HOL of two abstract machines, one functional (the SVM) and one imperative (the JVM). The latter is an extension of a previous one done by G. Klein [2]. We have also formalised the implementation of the first on top of the second, and defined a notion of equivalence between the abstract and the concrete states. Currently we are proving in Isabelle/HOL the main correctness theorem and its (many) auxiliary lemmas. The pay-off of this laborious job is to have a machine-checked certified implementation of our language Safe-Imp.

In [7] we showed a formal derivation from the source language operational semantics to the Safe-Imp semantics. Our next task will be to certify this translation. By joining the two certificates, we would hopefully have our functional language Safe correctly and proof-checked implemented on the Java Virtual Machine. This would be the basis for automatically generating formal certificates of some properties guaranteed by the Safe compiler such as termination, bounded space and absence of dangling pointers.

References

[1] G. Klein. *Verified Java Bytecode Verification.* PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[2] G. Klein and T. Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298:583–626, 2003.

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

[4] G. Klein, T. Nipkow, N. Schirmer, M. Strecker, and M. Wildmoser. Project VerifiCard. http://isabelle.in.tum.de/VerifiCard/, 2001–2003.

[5] T. Lindholm and F. Yellin. *The Java Virtual Machine Sepecification Second Edition.* The Java Series. Addison-Wesley, 1999.

[6] M. Montenegro, R. Peña, and C. Segura. An Inference Algorithm for Guaranteeing Safe Destruction. *Accepted for presentation in LOPSTR 2008.*

[7] M. Montenegro, R. Peña, and C. Segura. A Resource-Aware Semantics and Abstract Machine for a Functional Language with Explicit Deallocation. In *17th Int'l Workshop on Functional and (Constraint) Logic Programming, Siena, Italy July, 2008*, pages 1–15, 2008.

[8] 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, Valencia, Spain*, page to appear, July 2008.

[9] G. C. Necula. Proof-Carrying Code. In *Conference Record of POPL'97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM SIGACT and SIGPLAN, ACM Press, 1997.

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

[11] T. Nipkow, D. von Oheimb, and C. Push. Project Bali. http://isabelle.in.tum.de/Bali/, 1998–2000.

[12] R. Peña, C. Segura, and M. Montenegro. A Sharing Analysis for SAFE. In *Proceedings of the Seventh Symposium on Trends in Functional Programming, TFP'06*, pages 205–221, 2006.

[13] C. Pusch. Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle/HOL. In *TACAS'99*, pages 89–103. LNCS 1579, Springer, 1999.

[14] M. Wildmoser. Verified Proof Carrying Code. Ph.D. thesis, Institut für Informatik, Technical University Munchen, 2005.