A Certified Implementation on top of the Java Virtual Machine*

Javier de Dios Ricardo Peña

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
jdcastro@aventia.com, ricardo@sip.ucm.es

Abstract. Safe is a first-order functional language with unusual memory management features: memory can be both explicitly and implicitly deallocated at some specific points in the program text, and there is no need for a runtime garbage collector. The final code is bytecode of the Java Virtual Machine (JVM), so the language is useful for programming small devices based on this machine.
As an intermediate stage in the compiler's back-end, we have defined the *Safe Virtual Machine* (SVM), and have implemented this machine on top of the Java Virtual Machine (JVM). The paper presents the certified implementation of the SVM on top of the JVM. We have used the proof assistant Isabelle/HOL for this purpose.

1 Introduction

Safe[1] [20,15] was introduced as a research platform for investigating the suitability of functional languages for programming small devices and embedded systems with strict memory requirements. Its final aim is to be able to infer and certify —at compile time— safe upper bounds on memory consumption in a Proof Carrying Code environment [17]. Two features make *Safe* different from conventional functional languages: (1) Its region-based memory management system does not need a garbage collector; and (2) The programmer may ask for explicit destruction of memory cells, so that they could be reused by the program. The compiler produces as target language Java bytecode. These characteristics, together with the formal certification of memory safety properties, would make *Safe* useful for programming small devices.

Regions in *Safe* are inferred by the compiler and their allocation and deallocation are implicit. However, cell destruction, if desired, is explicit in the text and it is expressed as a special form of pattern matching. This is a dangerous feature which could result in having dangling pointers at runtime. The *Safe* compiler is at present equipped with a battery of static analyses, which taken as a whole infer the important property of absence of dangling pointers [20,15,14,16]. These

* Partially supported by the Spanish and Madrid Region Government grants S-0505/TIC/0407 (PROMESAS), and TIN2008-06622-C03-01/TIN (STAMP)

[1] http://dalila.sip.ucm.es/safe

analyses are conveyed on an intermediate language called *Core-Safe* (explained in Sec. 2.1), obtained after type-checking and desugaring the source language called *Full-Safe*. The back-end comprises two more phases:

1. A translation from *Core-Safe* to the bytecode language of an imperative abstract machine of our own, called the *Safe Virtual Machine* (SVM). This language is explained in Sec. 2.2.
2. A translation from SVM to the bytecode language of the *Java Virtual Machine* (JVM) [13].

We have decided to provide certificates on the absence of dangling pointers and (future certificates) on memory consumption at the *Core-Safe* level. The main reason for that is avoiding translating the certificates down to the JVM level, in parallel with the code. We also conjecture that this latter approach would result in huge certificates and huge checking times. For this reason, we prove instead that the translation does not destroy the certified properties. In particular, that the heap structure and the number of active cells are correctly mapped to the low level machine. Otherwise, absence of dangling pointers or memory consumption would not be preserved.

In a previous work [5], we certified the translation from *Core-Safe* to SVM. The main proof technique used there was structural induction on *Core-Safe* expressions. The distance between *Core-Safe* and the SVM was not so long as both languages shared the same heap definition, and the main emphasis was on proving that the resources 'consumed' at the *Core-Safe* level were the same that at the SVM level. Here we present the certification of the last translation step. The proof technique is different because here both languages are imperative. In essence we show that the JVM correctly simulates the SVM. The distance between both machines is so long (there is an expansion factor of around 20 between an SVM instruction and its translation to JVM) that the proofs are huge, although not difficult. The hardest part is showing that the final states in both machines preserve the simulation relation.

Machine-assisted compiler certification has been developed by several authors in the last few years. In Sec. 6 we review some of these works. As it is argued in [10,11], mechanised certification is superior to manual verification and of course to plain testing. For the certification being really trustable, the code running in the compiler's back-end should be *exactly the same* which has been proved correct by the proof-assistant. Fortunately, modern proof-assistants such as Coq [2] and Isabelle/HOL [19] provide code extraction facilities which deliver code written in some widely used languages such as Caml or Haskell. Of course, one must trust the translation done by the proof-assistant.

Isabelle/HOL is a well-known proof assistant, allowing to express definitions and properties in a formal language and to prove them with some human help. We have formalised in Isabelle/HOL the semantics of our abstract machine SVM, its translation to the JVM, and the JVM itself by extending a previous formalisation by G. Klein [7]. This was needed because Klein's machine was a rather small subset of the actual JVM and it did not cover some features needed by our implementation. This infrastructure allowed us to formally state and prove the correctness theorem.

The plan of the paper is as follows: In Section 2 we summarise our language *Safe* and formalise in Isabelle/HOL the semantics of the SVM; Section 3 presents the JVM formalisation made by Klein and our extension; in Section 4, we explain our design for mapping our machine to the JVM, and present the main code generation functions; the certification itself is summarised in Section 5: we define a simulation relation and prove that the pair formed by the instructions translation and our memory management system, correctly simulates the SVM semantics; there is finally a conclusions and related work section.

The paper is a summary of a rather large development whose full details can be found at `http://dalila.sip.ucm.es/safe/certifsvm2jvm`, where all the Isabelle/HOL theories containing the code generation functions, the lemmas, and the proofs are available.

2 The *Safe* language

Safe [20,15,14,16] is a first-order eager language with a syntax similar to Haskell's. Its runtime system uses *regions*, i.e. disjoint parts of the heap where the program allocates data structures. 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 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* 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, or for cell destruction) those regions where the arguments live, also (for reading, destruction, or insertion) the regions received as additional arguments, and the *self* working region. The region arguments are explicit in the intermediate code but not in the source, since 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 [14], the code is annotated with region arguments (those occurring after the `@`):

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

so that the tree is created in `treeSort`'s *self* region and deallocated upon termination. The destruction facilities are associated to pattern matching. For instance, we show here a constant space function appending two lists:

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

The ! mark is the way programmers indicate that the matched cell must be destroyed. 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.

2.1 *Core-Safe* and its translation to SVM

The *Safe* front-end desugars *Full-Safe* and produces a bare-bones functional language called *Core-Safe*. The transformation starts with region inference and continues with Hindler-Milner type inference, pattern matching desugaring into **case** expressions, transforming **where** clauses into **let** expressions, and some others. A *Core-Safe* program is a sequence of possibly recursive polymorphic data and function definitions followed by a main expression e whose value is the program result. Destructive pattern matching is transformed into **case!** expressions, and only constants or variables are allowed in function and constructor applications. Also, only variables are allowed in **case**/**case!** discriminants and in copy and reuse expressions. Region arguments are explicit in constructor and function applications and in the copy expression. Function definitions have additional region arguments $r_1, \ldots, r_m$ where the function is allowed to build data structures. As an example, we show the *Core-Safe* version of the above `append` function, and a main program invoking it:

$$
\begin{array}{l}
\mathit{append}\ \mathit{xs}\ \mathit{ys}\ @\ r = \textbf{case!}\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad \begin{array}{lcl}
[\,] & \rightarrow & \mathit{ys} \\
x : \mathit{xx} & \rightarrow & \textbf{let}\ \mathit{yy} = \mathit{append}\ \mathit{xx}\ \mathit{ys}\ @\ r\ \textbf{in} \\
 & & \textbf{let}\ \mathit{zz} = (x : \mathit{yy})\ @\ r\ \textbf{in}\ \mathit{zz};
\end{array} \\
\textbf{let}\ l = [\,]\ @\ \mathit{self}\ \ \textbf{in}\ \ \mathit{append}\ l\ l\ @\ \mathit{self}
\end{array}
$$

2.2 The Safe Virtual Machine

The *Safe* compiler translates *Core-Safe* into a set of sequences of imperative SVM instructions. These belong to the instruction set of the SVM, whose semantics in terms of configuration transitions is shown Fig. 1. 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 and k_0 are machine registers respectively denoting the topmost region in the heap and the topmost region that must be preserved upon reaching a normal form, S is the stack and cs is the code store where the instruction sequences are kept. For example, the *Core-Safe* `append` program of Sec. 2.1 generates the code store of Fig. 2.

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 b_i are its arguments. Regions are stacked as functions are invoked.

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

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

Fig. 1. The abstract machine SVM

Region identifiers j are natural numbers indicating the position of the region in the region stack. By $\mathit{\Delta}\mid_k$ we denote the heap obtained by deleting from $\mathit{\Delta}$ those regions above region 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 $\mathit{\Delta}$, or basic constants. 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 functions from *Core-Safe* to SVM here but, in order to understand the machine behaviour, we give some hints about it:

- **let** $x_1 = e_1$ **in** e_2 is translated into pushing a continuation for e_2 in the SVM stack (`PUSHCONT` instruction), followed by the instructions of e_1. If e_1 is a constructor application, a new cell is allocated for it (`BUILDCLS` instruction) and the execution proceeds with e_2.

$P_1 \mapsto [\mathit{BUILDCLS}\ \mathit{Nil}_0^2\ [\,]\ \mathit{self},\ \mathit{BUILDENV}\ [0, 0, \mathit{self}],\ \mathit{SLIDE}\ 3\ 1,\ \mathit{CALL}\ P_2]$
$P_2 \mapsto [\mathit{MATCH}!\ 0\ [P_3, P_4]]$
$P_3 \mapsto [\mathit{BUILDENV}\ [1],\ \mathit{SLIDE}\ 1\ 3,\ \mathit{DECREGION},\ \mathit{POPCONT}]$
$P_4 \mapsto [\mathit{PUSHCONT}\ P_5, \mathit{BUILDENV}\ [3,\ 5,\ 6], \mathit{SLIDE}\ 3\ 0, \mathit{CALL}\ P_2]$
$P_5 \mapsto [\mathit{BUILDCLS}\ \mathit{Cons}_1^2\ [1,\ 0]\ 5, \mathit{BUILDENV}\ [0], \mathit{SLIDE}\ 1\ 6, \mathit{DECREGION}, \mathit{POPCONT}]$

Fig. 2. Imperative code for the *Core-Safe* append program

- **case** expressions are translated into a MATCH/MATCH!/MATCHN instruction jumping to the appropriate alternative. Each one is a separate sequence.
- The translation of function application consists of pushing the arguments in the SVM stack (BUILDENV instruction), and then jumping to the function body (CALL instruction). Function calls are always tail recursive, so there is no need for a return instruction.
- When a normal form is reached: (1) the current environment is discarded 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).
- The copy $x@r$, reuse $x!$, and primitive operation $a_1 \oplus a_2$ expressions are respectively translated into sequences containing a COPY, a REUSE, and a PRIMOP instructions.

We now explain more closely each individual instruction: DECREGION deletes from the heap all the regions between the current region k and region k_0, excluding the latter; 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; PUSHCONT pushes a continuation represented by a region number and a code pointer.

Instruction COPY copies to region j the data structure starting at pointer b on top of the stack; REUSE creates a fresh pointer b' and makes it to point to the data structure pointed to by b on top of the stack; CALL jumps to a new instruction sequence and stacks an empty region $k + 1$; 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 matched cell constructor; MATCH! additionally destroys the matched cell; MATCHN is used when the **case** discriminant is a basic value. The equations respectively describe what happens when the discriminant is matched by one alternative, and when it is not matched and the default alternative must be taken.

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; BUILDCLS allocates fresh memory and constructs a heap cell. It receives a list of keys and the cell constructor C_r^m; SLIDE removes some parts of the stack and it is used to discerd environments when they are no longer needed.

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`}. It is introduced in Isabelle/HOL as an axiom, since it is needed for proving the correctness theorem.

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

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

The heap is modelled as a partial function from locations to pairs $(\mathit{Region}, \mathit{Cell})$, and a *nat* with the total number of regions. The stack is modelled as a list.

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

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, 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 $\mathit{CodeSequence} = \mathit{SafeInstr}\ \mathit{list}$
$\mathit{SVMCode} = (\mathit{CodeLabel} \times \mathit{CodeSequence} \times \mathit{FunName})\ \mathit{list}$
$\mathit{ContinuationMap} = \mathit{FunName} \rightharpoonup \mathit{CodeLabel}\ \mathit{list}$
$\mathit{CodeStore} = \mathit{SVMCode} \times \mathit{ContinuationMap}$

The program counter of the SVM is a pair $(\mathit{CodeLabel}, \mathit{nat})$ indicating the instruction sequence under execution and the next instruction of the sequence to be executed. The SVM state consists of the heap, 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 containing constructor static attributes needed at runtime, and a sizes table with sizes for the heap and the stack.

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

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

$\mathit{execSVM} :: \mathit{SafeImpProg} \Rightarrow \mathit{SVMState} \Rightarrow (\mathit{SVMState}, \mathit{SVMState})\ \mathit{Either}$
$\mathit{execSVM}\ ((\mathit{code}, \mathit{cm}), \mathit{ct}, \mathit{st})\ (h, k0, (l, i), S) =$
$\mathit{execSVMInst}\ (\mathit{the}\ (\mathit{map\_of}\ \mathit{code}\ l)\ !\ i)\ (\mathit{map\_of}\ \mathit{ct})\ h\ k0\ (l, i)\ S$

where *map_of*, defined in Isabelle/HOL, transforms a list of pairs into a partial function. If $\mathit{execSVM}\ P\ s$ gives $\mathit{Left}\ s$, this means that s is a stopping state. Otherwise, it gives $\mathit{Right}\ s'$. There is an equation defining *execSVMInst* for every SVM instruction. The definitions closely follow the semantics given in Fig. 1.

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

Fig. 3. Supported instructions of the JVM.

3 Formalisation of the JVM in Isabelle/HOL

In the past years, there have been some efforts to formally define the JVM in 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 [22] followed by Tobias Nipkow, Gerwin Klein and others in the framework of some EU-funded projects [9,6,8]. As starting point, we have used the definition of the JVM done in 2003 by G. Klein for Microjava, a subset of Java [7], and have extended it with a static heap and some instructions such as `Tableswitch`, `Invoke_static`, binary operators, and others. These extensions are part of the actual JVM and we needed in our implementation.

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

```
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)
      '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, 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 Fig. 3. They represent both a subset and an abstraction of the actual JVM instruction set [13].

The dynamic state of the JVM (*jvm_state*) is formed by four components: a possibly raised exception, a static heap (*sheap*), a dynamic heap (*dheap*), an initial heap, and a stack of frames. The second is a partial function from pairs

⟨class, field⟩ to values, and the third is a partial function from locations to either objects or arrays (*heap_entry*). An object contains its class name and a mapping from pairs ⟨field, class⟩ to values. An array consists of its elements type, its length, and a partial mapping from indices to values. A frame (*frame*) is formalised as a tuple containing the operand stack, the local variables (these include the *this* pointer and the method arguments), the class name, the method signature, the program counter, and a tag. This and the initial heap are not present in the actual JVM, and they are related to a proof about the bytecode type system made by Klein.

datatype $\mathit{heap\_entry} = \ \ \mathit{Obj}\ \mathit{cname}\ (\mathit{vname} \times \mathit{cname} \rightharpoonup \mathit{val})$
$\qquad\qquad\qquad\qquad\ \mid \mathit{Arr}\ \mathit{ty}\ \mathit{nat}\ (\mathit{nat} \rightharpoonup \mathit{val})$

types $\mathit{sheap} = \mathit{cname} \times \mathit{vname} \rightharpoonup \mathit{val}$ -- static heap
$\qquad\ \ \mathit{dheap} = \mathit{loc} \rightharpoonup \mathit{heap\_entry}$ -- dynamic heap
$\qquad\ \ \mathit{frame} = \mathit{opstack} \times \mathit{locvars} \times \mathit{cname} \times \mathit{sig} \times \mathit{pc} \times \mathit{tag}$
$\qquad\ \ \mathit{jvm\_state} = \mathit{val}\ \mathit{option} \times \mathit{sheap} \times \mathit{dheap} \times \mathit{ini\_heap} \times \mathit{frame}\ \mathit{list}$

Klein defines a function $\mathit{exec} :: \mathit{jvm\_prog} \times \mathit{jvm\_state} \Rightarrow \mathit{jvm\_state}\ \mathit{option}$, executing the next instruction in the machine, which we have extended to the added instructions. In addition, he defines the reflexive-transitive closure of the $\mathit{exec}\ P$ relation, for a given program P, as follows:

$\mathit{exec\_all} :: [\mathit{jvm\_prog}, \mathit{jvm\_state}, \mathit{jvm\_state}] \Rightarrow \mathit{bool} \qquad (\_ \vdash \_ - \mathit{jvm} \rightarrow \_\ )$
$P \vdash s - \mathit{jvm} \rightarrow t \equiv (s,t) \in \{(s,t)\ .\ \mathit{exec}\ (P,s) = \mathit{Some}\ t\}^*$

4 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 collects unused objects. On the contrary, the SVM has explicit 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 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.

4.1 Mapping the SVM data structures

As explained in Sec. 1, one aim of Safe is to statically infer and certify upper bounds for heap and stack sizes. In order to make the reusing of 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.

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. 4. The interface of the classes `Heap` and `CellFactory`.

The heap is implemented by two classes, `DirectoryCell` 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 Fig. 4. 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 and a static field k, so that constant time access to each region is provided. Register k_0 is also a static field of the class `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 moving 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 consisting of the classes `Heap, DirectoryCell`, and `Stack`.

4.2 Mapping the SVM code

The code generated by translating a SVM program consists of a single JVM class `PSafe` with a single method `PSafeMain()`. *Core-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 to RTS methods are allowed in the sequence.

A SVM `MATCH` instruction is implemented by using the JVM `Tableswitch` instruction, 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 $0, \dots, totC - 1$, being $totC$ the total number of continuations in the program, which in turn is equal to the number of its non constructor-building **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.

4.3 Translation functions

As we have said, we have defined in Isabelle the translation from SVM to JVM and used its code extraction facilities to produce the Haskell code actually executed in the compiler. The translation provides, as add-ons, a function mapping the SVM program counters to the JVM program counters corresponding to the translation, a function mapping continuation labels to the small integers mentioned in Sec. 4.2, and a function assigning to each constructor a unique number. These mappings are needed later to define the simulation relation between the states of the SVM and the JVM.

$$\begin{array}{l} \mathit{codeMap} :: \mathit{PC} \rightharpoonup \mathit{pc} \\ \mathit{contMap} :: \mathit{CodeLabel} \rightharpoonup \mathit{nat} \\ \mathit{consMap} :: \mathit{Constructor} \rightharpoonup \mathit{nat} \\ \mathit{trSVM2JVM} :: \mathit{SafeImpProg} \Rightarrow \mathit{jvm\_prog} \times \mathit{codeMap} \times \mathit{contMap} \times \mathit{consMap} \end{array}$$

This function creates the initialisation code and the constructor mapping, builds the `SafeMain` class and attaches to it the RTS classes. To produce the bytecode of the only method of that class, it uses the function

$$\begin{array}{l} \mathit{trCodeStore} :: [\mathit{CodeLabel}, \mathit{pc}, \mathit{ContinuationMap}, \mathit{consMap}, \mathit{SVMCode}] \\ \qquad\qquad\quad \Rightarrow \mathit{instrlist} \ \times \mathit{codeMap} \times \mathit{contMap} \end{array}$$

which traverses the SVM code sequences, accumulating the bytecode fragments and the program counter mapping produced by them. It also assigns unique numbers to continuations. In order to translate a sequence, it uses the function

$$\begin{array}{l} \mathit{trSeq} :: [\mathit{contMap}, \mathit{consMap}, \mathit{pc}, \mathit{pc} \times \mathit{codeMap}, \mathit{CodeLabel} \times \mathit{CodeSequence} \times \mathit{FunName}] \\ \qquad\quad \Rightarrow (\mathit{pc} \times \mathit{codeMap}) \times \mathit{instr\ list} \end{array}$$

which traverses one SVM sequence, accumulating the bytecode fragments produced by each SVM instruction. It also updates the program counter mapping after each translation. The main translation function, defined by cases on the SVM instruction being translated is

$$\mathit{trInstr} :: [\mathit{pc}, \mathit{codeMap}, \mathit{contMap}, \mathit{consMap}, \mathit{pc}, \mathit{SafeInstr}] \Rightarrow \mathit{instr\ list}$$

where the first pc is the one corresponding to the first JVM instruction of the translation, and the second one is the program counter of the global `Tableswitch` mentioned in Sec, 4.2 to deal with continuations. As an example, we show in Fig. 5 the bytecode resulting from the translation of `POPCONT`. The code of all these functions can be found at `http://dalila.sip.ucm.es/safe/certifsvm2jvm`.

```
[Getstatic Sf stackC,
 Getstatic topf stackC,
 Dup2,
 Dup2,
 Dup2,
 ArrLoad,
 Store 1,        (* local1 <- b *)
 LitPush (Intg 1),
 BinOp Substract,
 ArrLoad,
 Store 2,        (* local2 <- k’ *)
 LitPush (Intg 2),
 BinOp Substract,
 ArrLoad,
 Store 3,              (* local3 <- p *)
 LitPush (Intg 2),
 BinOp Substract,
 Dup,
 Putstatic topf stackC, (* top <- top - 2 *)
 Load 1,
 ArrStore,             (* S[top] <- b *)
 Load 2,
 Putstatic k0f heapC,   (* k0 <- k’ *)
 Load 3,               (* jump to continuation *)
 Goto (trAddr pcc (pc + incPop))]
```

Fig. 5. The JVM bytecode produced by `POPCONT`

5 Certification of the implementation

The main idea of the proof is defining a simulation relation between SVM and JVM states and showing that both machines evolve through states made equivalent by the relation when executing a SVM program and its translation.

To define the simulation relation we must consider that part of the JVM state is implemented by the static data structures kept in the RTS classes `Heap` and `Stack`, and that the rest is kept in the cell objects and arrays of the dynamic heap. The relation admits a SVM state to be simulated by several JVM states, since there is an abstraction when going from lists of cells in the JVM to set of cells in the SVM. The critical part of the relation is the existence of a bijection between the SVM heap locations and the JVM dynamic heap locations corresponding to active cells, i.e. cells linked in some list of the region stack. The bijection must preserve the heap structure in the sense that equivalent cells must point to equivalent cells. In the following, we will assume that P is a SVM program and $(P', \mathit{cdm}, \mathit{ctm}, \mathit{com}) = \mathit{trSVM2JVM}\ P$ its translation.

Definition 1. Given an injection $g :: \mathit{dom}\ H \Rightarrow \mathit{loc}$, and a constructor mapping com, the JVM dynamic heap h and the region stack regS with k' regions *simulate* the SVM heap (H, k), denoted $\mathit{equivH}\ (H, k)\ h\ k' \mathit{com}\ \mathit{regS}\ g$, if:

$$\mathit{range}\ g = \mathit{activeCells}\ \mathit{regS}\ k' \wedge k = k' \wedge \forall l \in \mathit{dom}\ H\,.\,\mathit{equivC}\ (H\ l)\ (h\ (g\ l))\ \mathit{com}\ g$$

The first condition guarantees that g is in fact a bijection. Predicate equivC (not shown) defines that two cells are equivalent under g and com when both live in the same region j and contain pointers made equivalent by g in equivalent argument positions. Equivalent constructor names are the string C in the SVM and the unique number $\mathit{com}\ C$ in the JVM.

Definition 2. The JVM state $s' = (\mathit{None}, h_s, h_d, h_i, ([], \mathit{vs}, \text{“PSafe”}, (\text{“PSafeMain”}, \mathit{ts}), \mathit{pc}, \mathit{tag})\#[])$ simulates the SVM state $s = (H, k_0, PC, S)$, denoted $\mathit{cdm}, \mathit{ctm}, \mathit{com} \vdash s \triangleq s'$, if there exists an injection $g :: \mathit{dom}\ H \Rightarrow \mathit{loc}$ such that:

1. $\mathit{equivH}\ H\ h_d\ k' \mathit{com}\ \mathit{regS}\ g$, where the region stack (regS, k') is obtained from h_s and h_d by using the RTS class `Heap`.

2. *equivS* S S' *top ctm g*, where the stack (S', top) is obtained from h_s and h_d by using the RTS class `Stack`.
3. $k_0 = k'_0$, where k'_o is the static field k_0 of the class `Heap`.
4. $pc = cdm\ PC$.

Predicate *equivS* (not shown) defines that the JVM stack (S', top) simulates the SVM stack S when, position by position, both contain either the same two basic values, or two heap locations made equivalent by g, or two continuations made equivalent by ctm[2].

Notice that the simulation relation guarantees that the heap structure is exactly the same in both machines. So, properties such as the number of active cells and the absence of dangling pointers are preserved.

The main correctness 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. The Isabelle/HOL formalisation is:

```
theorem correctSVM2JVM :
  ⟦ (P',cdm,ctm,com) = trSVM2JVM P;
    cdm, ctm, com ⊢ S1 ≜ S1';
    execSVM P S1 = Right S2 ⟧ ⟹
      ∃ S2' . P' ⊢ S1' -jvm→ S2' ∧ cdm, ctm, com ⊢ S2 ≜ S2'
```

A first set of lemmas deal with the static properties of the translation and prove that, if (p, i) and pc are two program counters made equivalent by cdm, then the JVM bytecode starting at pc is exactly the translation of the SVM instruction found at (p, i). The topmost one is the following:

```
lemma fun_SVM2JVM [rule_format]:
    (P', cdm, ctm, com) = trSVM2JVM ((svms, ctmap), ini, ct, st) ⟶
    l < length svms ⟶
    svms ! l = (p,seq,fn) ⟶
    i < length seq ⟶
    svm = fst (the (map_of svms p)) ! i ⟶
    pc = the (cdm (p,i)) ⟶
    bytecode = extractBytecode P' ⟶
    (∃ cdm' ctm' pcc inss bytecode' .  inss = trInstr pc cdm' ctm' com pcc svm ∧
                                       drop pc bytecode = inss @ bytecode')
```

After some initial massaging, the kernel of the main proof is done by cases on the instruction executed in the SVM. We have proved one auxiliary lemma for each SVM instruction. We show below the one corresponding to `POPCONT`:

```
lemma execSVMInstr_POPCONT :
  ⟦ (P', cdm, ctm, com) = trSVM2JVM ((svms, ctmap), ini, ct, ah, ai, bc);
    cdm , ctm, com ⊢ ((hm, k), k0, (l, i), S) ≜ S1';
    (fst (the (map_of svms l)) ! i) = POPCONT;
```

[2] More details can be found at `http://dalila.sip.ucm.es/safe/certifsvm2jvm`.

$$\begin{array}{l}
\quad \mathit{execSVMInst\ POPCONT\ (map\_of\ ct)\ (hm,\ k)\ k0\ (l,\ i)\ S = Right\ S2;} \\
\quad \mathit{drop\ (the\ (cdm\ (l,\ i)))\ (extractBytecode\ P') =} \\
\quad \mathit{trInstr\ (the\ (cdm\ (l,\ i)))\ cdm'\ ctm'\ com\ pcc\ POPCONT\ @\ bytecode'} \\
]\!] \Longrightarrow\ \ \exists\ \mathit{v'\ sh'\ dh'\ ih'\ fms'}\ .\ \ \mathit{P'} \vdash\ \ \mathit{S1'\ \text{-}jvm\!\rightarrow\ (v',sh',dh',ih',\ fms')} \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{cdm\ ,\ ctm,\ com}\ \vdash\ \ \mathit{S2}\ \triangleq\ \mathit{(v',sh',dh',ih',\ fms')}
\end{array}$$

The conclusion of the lemma is the same as that of the main theorem, but the premises inform us that the instructions about to be executed in the JVM are exactly those produced by the translation of POPCONT. The proof of this kind of lemmas is rather long and consists of passing through all the intermediate JVM states determined by the JVM bytecode and showing that the final state is equivalent to the arrival state in the SVM. If the bytecode contains loops, the proof become harder as we must introduce invariants and prove loop termination.

6 Conclusions and Related Work

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 [7]. We have also formalised the implementation of the first on top of the second, and defined a simulation relation between the abstract and the concrete states. As part of the relation, we have proved the existence of a bijection across the execution, guaranteeing that the number of cells and the heap structure is the same in both machines. In a previous work, we proved that a similar equivalence held between the *Core-Safe* and the SVM levels of the translation. Considering both proofs as a whole, this certifies that the memory consumption and the absence of dangling pointers properties certified at the *Core-Safe* level are preserved in the JVM code actually executed.

The complete specification in Isabelle/HOL of the syntax and semantics of both languages, of the translation functions, the theorems and the proofs, represent about one person-year of effort. Including comments, about 21 000 lines of Isabelle/HOL scripts have been written, and about 120 lemmas, some of them very long, have been proved. Isabelle/HOL features a Higher-Order Logic and gives enough facilities for defining recursive and higher-order functions. These are written in much the same way as a programmer would do in a modern functional language such as ML or Haskell. Isabelle/HOL provides also inductive predicates, inductive n-relations, transitive closures as well as ordinary first-order logic. This has made it easy to express the desired properties with almost the same concepts one would use in hand-written proofs. Partial functions have also been very useful in modelling programming language structures such as environments, heaps, and the like. Being able to quantify these objects in HOL has been essential for stating and proving the theorems.

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 at [4]. 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. For instance, [18] uses a constraint solver to asses the validity of the GNU C compiler translations. They do not try to prove the compiler correctness but to *validate its output*, by comparing it with the corresponding input. This technique was originally proposed in [21]. A more recent experiment in compiler validation is [12], where the source is the term language of HOL and the target is assembly language of the ARM processor.

More closely related to our work is [23] where the author 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. He defines a big-step semantics for μ-Java and a sate-transition semantics for the small JVM (17 bytecode instructions). Then, the translation functions are defined and a correctness theorem similar to ours is proved. This work can be considered as a first attempt, and it was considerably extended by Klein, Nipkow, Berghofer, and Strecker himself in [7,8,1]. Only the latter claims that the extraction facilities of Isabelle/HOL have been used to produce an actually running Java compiler. The main emphasis is on formalisation of Java and JVM features and on creating an infrastructure on which other authors could verify properties of Java or Java bytecode programs.

A realistic C compiler for programming embedded systems has been built and verified in [3,10,11]. 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. They provide figures witnessing that the compile times obtained are competitive with those of *gcc* running with level-2 optimisations activated. This is perhaps the biggest project on machine-assisted compiler verification done up to now.

As we have said in Sec. 1, the motivation for verifying the *Safe* back-end arises in a different context. We have approached this development because we found it more rapid than translating the *Core-Safe* properties to certificates at the level of the JVM. Also, we expected the size of our certificates to be considerably smaller than the ones obtained with the other approach. Additionally to previous efforts, we have complemented functional correctness with a proof of resource consumption and memory structure preservation.

Acknowledgement We are grateful to Delfin Rupérez for providing preliminary Isabelle/HOL code for the RTS, the JVM extensions, and the translation.

References

1. S. Berghofer and M. Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. In *Proc. Compiler Optimization Meets Compiler Verification, COCV'03*, pages 33–50. ENTCS, 2003.
2. Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

3. S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *Symp. on Formal Methods, FM'06*, LNCS 4085, pages 460–475. Springer, 2006.
4. M. A. Dave. Compiler verification: a bibliography. *SIGSOFT Software Engineering Notes*, 28(6):2–2, 2003.
5. J. de Dios and R. Peña. Formal Certification of a Resource-Aware Language Implementation. In *Int. Conf. on Theorem Proving in Higher Order Logics, TPHOL'09, Munich (Germany)*, pages 1–15. LNCS 5674 (to appear), Springer, August 2009.
6. G. Klein. *Verified Java Bytecode Verification.* PhD thesis, Institut für Informatik, Technische Universität München, 2003.
7. G. Klein and T. Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298:583–626, 2003.
8. 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.
9. G. Klein, T. Nipkow, N. Schirmer, M. Strecker, and M. Wildmoser. Project VeriFiCard. http://isabelle.in.tum.de/VerifiCard/, 2001–2003.
10. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Principles of Programming Languages, POPL'06*, pages 42–54. ACM Press, 2006.
11. X. Leroy. A formally verified compiler back-end. Submitted, 79 pags., July 2008.
12. G. Li, S. Owens, and K. Slind. Structure of a Proof-Producing Compiler for a Subset of Higher Order Logic. In *European Symp. on Programming, ESOP'07*, pages 205–219. LNCS 4421, Springer, 2007.
13. T. Lindholm and F. Yellin. *The Java Virtual Machine Sepecification Second Edition.* The Java Series. Addison-Wesley, 1999.
14. M. Montenegro, R. Peña, and C. Segura. A Simple Region Inference Algorithm for a First-Order Functional Language. In *Trends in Functional Programming, TFP'08, Nijmegen (The Netherlands)*, pages 194–208, May 2008.
15. M. Montenegro, R. Peña, and C. Segura. A Type System for Safe Memory Management and its Proof of Correctness. In *ACM Principles and Practice of Declarative Programming, PPDP'08, Valencia, Spain, July. 2008*, pages 152–162, 2008.
16. M. Montenegro, R. Peña, and C. Segura. An Inference Algorithm for Guaranteeing Safe Destruction. In *Selected papers of Logic-Based Program Sinthesis and Transformation, LOPSTR'08, LNCS 5438, Springer.*, pages 135–151, 2009.
17. G. C. Necula. Proof-Carrying Code. In *ACM SIGPLAN-SIGACT Principles of Programming Languages, POPL'97*, pages 106–119. ACM Press, 1997.
18. G. C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Notices*, 35(5):83–94, 2000.
19. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic.* LNCS 2283. Springer, 2002.
20. R. Peña, C. Segura, and M. Montenegro. A Sharing Analysis for SAFE. In *Selected Papers of the 7th Symp. on Trends in Functional Programming, TFP'06*, pages 109–128. Intellect, 2007.
21. A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS'98*, LNCS 1384, Springer, pages 151–166, 1998.
22. C. Pusch. Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle/HOL. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS'99*, pages 89–103. LNCS 1579, Springer, 1999.
23. M. Strecker. Formal Verification of a Java Compiler in Isabelle. In *Conference on Automated Deduction, CADE'02*, LNCS 2392, Springer, pages 63–77, 2002.