Formal Certification of a Resource-Aware Language Implementation*

Javier de Dios and Ricardo Peña

Universidad Complutense de Madrid, Spain
C/ Prof. José García Santesmases s/n. 28040 Madrid
Tel.: 91 394 7627; Fax: 91 394 7529
`jdcastro@aventia.com, ricardo@sip.ucm.es`

Abstract. The paper presents the development, by using the proof assistant Isabelle/HOL, of a compiler back-end translating from a functional source language to the bytecode language of an abstract machine. The Haskell code of the compiler is extracted from the Isabelle/HOL specification and this tool is also used for proving the correctness of the implementation. The main correctness theorem not only ensures functional semantics preservation but also resource consumption preservation: the heap and stacks figures predicted by the semantics are confirmed in the translation to the abstract machine.

The language and the development belong to a wider Proof Carrying Code framework in which formal compiler-generated certificates about memory consumption are sought for.

Keywords: compiler verification, functional languages, memory management.

1 Introduction

The first-order functional language *Safe* has been developed in the last few years as a research platform for analysing and formally certifying two properties of programs related to memory management: absence of dangling pointers and having an upper bound to memory consumption.

Two features make *Safe* different from conventional functional languages: (a) the memory management system does not need a garbage collector; and (b) the programmer may ask for explicit destruction of memory cells, so that they could be reused by the program. These characteristics, together with the above certified properties, make *Safe* useful for programming small devices where memory requirements are rather strict and where garbage collectors are a burden both in space and in service availability.

The *Safe* compiler is equipped with a battery of static analyses which infer such properties [15,16,17,22]. These analyses are carried out 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). We call this bytecode language *Safe-Imp* and it is explained in Sec. 2.4.

* Work partially funded by the projects TIN2008-06622-C03-01/TIN (STAMP), and S-0505/ TIC/ 0407 (PROMESAS).

2. A translation from *Safe-Imp* to the bytecode language of the *Java Virtual Machine* (JVM) [13].

We have proved our analyses correct and are currently generating Isabelle/HOL [20] scripts which, given a *Core-Safe* program and the annotations produced by the analyses, will mechanically certify that the program satisfies the properties inferred by the analyses. The idea we are trying to implement, consistently with the *Proof Carrying Code* (PCC) paradigm [18], is sending the code generated by the compiler together with the Isabelle/HOL scripts to a hypothetical code consumer who, using another Isabelle/HOL system and a database of previously proved theorems, will check the property and consequently trust the code. The annotations consist of special types in the case of the absence of dangling pointers property, and will consist of some polynomials when the space consumption analysis is finished. At this point of the development we were confronted with two alternatives:

- Either to translate the properties obtained at *Core-Safe* level to the level of the JVM bytecode, by following for instance some of the ideas of [2].
- Or to provide the certificates at the *Core-Safe* level. Then the consumer should trust that our back-end does not destroy the *Core-Safe* properties, or better, we should provide evidence that these properties are preserved.

The first alternative was not very appealing in our case. Differently to [2], where the certificate transformation is carried on at the same intermediate language, here the distance between our *Core-Safe* language and the target language is very large: the first one is functional and the second one is a kind of assembly language; new structures such as the frames stack, the operand stack, or the program counter are present in the second but not in the first; we have built a complete memory management runtime system on top of the JVM in order to avoid its built-in garbage collector, etc. The translated certificate should provide invariants and properties for all these structures. Even if all this work were done, the size of the certificates and the time needed to check them would very probably be huge. The figures reported in [26] for JVM bytecode-level certificates seem to confirm this assertion.

The second alternative has other drawbacks. One of them is that the *Core-Safe* program must be part of the certificate, because the consumer must be able to relate the properties stated at source level with the low-level code being executed. Providing the source code is not allowed in some PCC scenarios. The second drawback is that the back-end should be formally verified, and both the translation algorithm, and the theorem proving its correctness must be in the consumer database. We have chosen this second alternative because smaller certificates can be expected, but also because we feel that proving the translation correct once for all programs is more reasonable in our case than checking this correctness again and again for every translated program.

Machine-assisted compiler certification has been developed by several authors in the last few years. In Sec. 6 we review some of these works. For the certification being really trusty, 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 [4] and Isabelle/HOL provide code extraction facilities which deliver code written in some wider-use languages such as Caml or Haskell. Of course, one must trust the translation done by the proof-assistant.

In this paper we present the certification of the first pass explained above (*Core-Safe* to *Safe-Imp*). The second pass (*Safe-Imp* to JVM bytecode) is currently being completed. The reader can find a preliminary version of it in [21].

The main improvement of this work with respect to previous efforts in compiler certification is that we prove, not only the preservation of functional semantics, but also the preservation of the resource consumption properties. As it is asserted in [11], this property can be lost as a consequence of some compiler optimisations. For instance, some auxiliary variables not present in the source may appear during the translation. In our framework, it is essential that memory consumption is preserved during the translation, since we are trying to certify exactly this property. To this aim, we introduce at *Core-Safe* level a *resource-aware* semantics and then prove that this semantics is preserved in the translation to the abstract machine.

With the aim of facilitating the understanding of the paper, and also avoiding descending to many low level details, we have made available the Isabelle/HOL scripts at `http://dalila.sip.ucm.es/safe/theories`. We recommend the reader to consult this site while reading in order to match the concepts described here with its definition in Isabelle/HOL. The paper is structured as follows: after this introduction, in Sec. 2 we motivate our *Safe* language and then present the syntax and semantics of the source and target languages. Then, Sec. 3 explains the translation and gives a small example of the generated code. Sections 2 and 3 contain large portions of material already published in [14,16]. We felt that this material was needed in order to understand the certification process. Sec. 4 is devoted to explaining the main correctness theorem and a number of auxiliary predicates and relations needed in order to state it. Sec. 5 summarises the lessons learnt, and finally a Related Work section closes the paper.

2 The Source and Target Languages

2.1 Full-Safe and Core-Safe

Safe is a first-order polymorphic functional language with a syntax similar to that of (first-order) Haskell, and with some facilities to manage memory. The memory model is based on heap regions where data structures are built. A region is a collection of cells and a cell stores exactly one constructor application. However, in *Full-Safe* regions are implicit. These are inferred [15] when *Full-Safe* is desugared into *Core-Safe*. The allocation and deallocation of regions are bound to function invocations: a *working region* is allocated when entering the call and deallocated when exiting it. All data structures allocated in this region are lost.

Inside a function, data structures may be built but they can also be destroyed by using a *destructive pattern matching*, denoted by the symbol !, which deallocates the cell corresponding to the outermost constructor. Using recursion the recursive spine of the whole data structure may be deallocated. As an example, we show an append function destroying the first list's spine, while keeping its elements in order to build the result:

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

This appending needs constant (in fact, zero) additional heap space, while the usual version needs linear additional heap space. The fact that the first list is lost

prog	$\rightarrow \overline{\mathit{data}_i}^{n} ; \overline{\mathit{dec}_j}^{m} ; e$	$\{$*Core-Safe* program$\}$
data	$\rightarrow \textbf{data}\ T\ \overline{\alpha_i}^{n}\ @\ \overline{\rho_j}^{m} = \overline{C_k\ \overline{t_{ks}}^{n_k}\ @\ \rho_m}^{l}$	$\{$recursive, polymorphic data type$\}$
dec	$\rightarrow\ f\ \overline{x_i}^{n}\ @\ \overline{r_j}^{l} = e$	$\{$recursive, polymorphic function$\}$
e	$\rightarrow\ a$	$\{$atom: literal c or variable $x\}$
	$\mid x\ @\ r$	$\{$copy data structure x into region $r\}$
	$\mid x!$	$\{$reuse data structure $x\}$
	$\mid a_1 \oplus a_2$	$\{$primitive operator application$\}$
	$\mid f\ \overline{a_i}^{n}\ @\ \overline{r_j}^{l}$	$\{$function application$\}$
	$\mid \textbf{let}\ x_1 = \mathit{be}\ \textbf{in}\ e$	$\{$non-recursive, monomorphic$\}$
	$\mid \textbf{case}\ x\ \textbf{of}\ \overline{\mathit{alt}_i}^{n}$	$\{$read-only case$\}$
	$\mid \textbf{case}!\ x\ \textbf{of}\ \overline{\mathit{alt}_i}^{n}$	$\{$destructive case$\}$
alt	$\rightarrow C\ \overline{x_i}^{n} \rightarrow e$	$\{$case alternative$\}$
be	$\rightarrow\ C\ \overline{a_i}^{n}\ @\ r$	$\{$constructor application$\}$
	$\mid e$	

Fig. 1. *Core-Safe* syntax

is reflected, by using the symbol ! in the type inferred for the function $\mathit{appendD} :: \forall a \rho_1 \rho_2 \,.\, [a]!@\rho_1 \rightarrow [a]@\rho_2 \rightarrow \rho_2 \rightarrow [a]@\rho_2$, where ρ_1 and ρ_2 are polymorphic types denoting the regions where the input and output lists should live. In this case, due to the sharing between the second list and the result, these latter lists should live in the same region. Another possibility is to destroy part of a data structure and to *reuse* the rest in the result, as in the following destructive *split* function:

```
splitD 0 zs!      = ([], zs!)
splitD n []!      = ([], [])
splitD n (y:ys)!  = (y:ys1, ys2) where (ys1, ys2) = splitD (n-1) ys
```

The righthand side $zs!$ expresses reusing the remaining list. The inferred type is:

$$\mathit{splitD} :: \forall a \rho_1 \rho_2 \rho_3 \,.\, \mathit{Int} \rightarrow [a]!@\rho_2 \rightarrow \rho_1 \rightarrow \rho_2 \rightarrow \rho_3 \rightarrow ([a]@\rho_1, [a]@\rho_2)@\rho_3$$

Notice that the regions used to build the result appear as additional arguments. The data structures which are not part of the function's result are inferred to be built in the local working region, which we call *self*, and they die at function termination. As an example, the tuples produced by the internal calls to `splitD` are allocated in their respective *self* regions and do not consume memory in the caller regions. The type of these internal calls is $\mathit{Int} \rightarrow [a]!@\rho_2 \rightarrow \rho_1 \rightarrow \rho_2 \rightarrow \rho_{\mathit{self}} \rightarrow ([a]@\rho_1, [a]@\rho_2)@\rho_{\mathit{self}}$, which is different from the external type because we allow polymorphic recursion on region types. More information about *Safe* and its type system can be found at [16].

The *Safe* front-end desugars *Full-Safe* and produces a bare-bones functional language called *Core-Safe*. The transformation starts with region inference and follows with Hindley-Milner type inference, desugaring pattern matching into **case** expressions, **where** clauses into **let** expressions, collapsing several function-defining equations into a single one, and some other transformations.

In Fig. 1 we show *Core-Safe*'s syntax, which is defined in Isabelle/HOL as a collection of datatypes. A program *prog* is a sequence of possibly recursive polymorphic data and function definitions followed by a main expression e 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. Constructor applications are only allowed in **let** bindings. Only atoms are used in applications, and only variables are used in **case**/**case!** discriminants, copy and reuse expressions. Region arguments are explicit in constructor and function applications and in the copy expression. Function definitions have additional region arguments $\overline{r_j}^l$ where the function is allowed to build data structures. In the function's body only the r_j and its working region *self* may be used.

2.2 Core-Safe Semantics

In Figure 2 we show the resource-aware big-step semantics of *Core-Safe* expressions. A judgement of the form $E \vdash h, k, td, e \Downarrow h', k, v, r$ means that the expression e is successfully 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. Arguments k can be considered as attributes of their respective heaps. We highlight them in order to emphasise that the evaluation starts and ends with the same number of regions, and also to show when regions are allocated and deallocated. A value v is either a constant or a heap pointer. The argument td and the result r have to do with resource consumption and will be explained later. The semantics can be understood disregarding them. Moreover, forgetting about resource consumption produces a valid value semantics for the language.

A runtime environment E maps program variables to values and region variables to actual region identifiers which consist of natural numbers. As region allocation/deallocation are done at function invocation/return time, the live regions are organised in a region stack. A region identifier is just its offset from the bottom of this stack. We adopt the convention that for all E, if c is a con-

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

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

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

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

$$\frac{c = c_1 \oplus c_2}{E[a_1 \mapsto c_1, a_2 \mapsto c_2] \vdash h, k, td, a_1 \oplus a_2 \Downarrow h, k, c, ([\,]_k, 0, 2)} \ [\mathit{Primop}]$$

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

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

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

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

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

Fig. 2. Resource-Aware Big-Step Operational Semantics of *Core-Safe* expressions

stant, then $E(c) = c$. A heap h is a finite mapping from fresh variables p to constructor cells w of the form $(j, C\,\overline{v_i}^n)$, meaning that the cell resides in region j. By $h[p \mapsto w]$ we denote a heap h where the binding $[p \mapsto w]$ is highlighted, while $h \uplus [p \mapsto w]$ denotes 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 runtime system *copy* function copying the recursive part of the data structure pointed to by p, and living in a region j', into a (possibly different) region j. 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 cells in this region or pass this region as an argument to other functions. Before returning from the function, all cells created in region $k + 1$ are deleted. 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 semantics is defined in Isabelle/HOL as an inductive relation. The environment E is split into a pair (E_1, E_2) separating program variables from region arguments bindings. These and the heap are modelled as partial functions. Even though all functions are total in Isabelle/HOL, a partial function, denoted $'a \rightharpoonup\ 'b$, can be easily defined as the total function $'a \Rightarrow\ 'b\ option$, where $f\ x = None$ represents that f is not defined at x.

2.3 Resource Consumption

The semantics relates the evaluation of an expression e to a resource vector $r = (\delta, m, s)$ obtained as a side effect. The first component is a partial function $\delta : \mathbb{N} \rightarrow \mathbb{Z}$ giving for each region k in scope 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. The additional argument *td* is the number of bindings in E which can be discarded when a normal form is reached or at function invocation. It coincides with the value returned by the function *topDepth* of Sec. 3. As we will see there, the runtime environment E is kept in the evaluation stack and (part of) this environment is discarded by the abstract machine in those situations. By $[\,]_k$ we denote the function $\lambda n.0$ if $0 \leq n \leq k$, and $\lambda n.\bot$ otherwise. By $\delta_1 + \delta_2$ we denote 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 argument 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 of these resource consumption figures takes into account how the translation will transform, and the abstract machine will execute, the corresponding expression. For instance, in rule *App* the number $\max\{n+l, s+n+l-td\}$ of fresh stack words takes into account that the first $n+l$ words are needed to store the actual arguments in the stack, then the current environment of length td is discarded, and then the function body is evaluated. In rule Let_1, a *continuation* (2 words, see Sec. 2.4) is stacked before evaluating e_1, and this evaluation leaves a value in the stack before evaluating e_2. Hence, the computation $\max\{2+s_1, 1+s_2\}$.

2.4 Safe-Imp Syntax and Semantics

Safe-Imp is the bytecode language of the SVM. Its syntax and semantics is depicted in Figure 3. A configuration of the SVM consists of the five components $(is, (h,k), k_0, S, cs)$, where is is the current instruction sequence, (h,k) is the current heap, k being its topmost region, S is a stack and cs is the code store where the instruction sequences resulting from the compilation of program fragments are kept. A code store is a partial function from *code labels*, denoted $p, q, \ldots$, to bytecode lists. The component k_0 is a low watermark in the heap registering which one must be the topmost region when a normal form is reached (see the semantics of the *DECREGION* instruction). The property $k_0 \leq k$ is an invariant of the execution. By $b, b_i, \ldots$ we denote heap pointers or any other item stored in the stack. The stack contains three kinds of objects: values, regions and continuations.

$$\begin{array}{lll} so & \rightarrow v \mid j \mid (k,p) & \{\text{stack object}\} \\ S & \rightarrow so\ list & \{\text{stack}\} \end{array}$$

The semantics of the *Safe-Imp* instructions is shown 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, and by $S!j$, the j-th element of the stack S counting from the top and starting at 0. A more complete view on how this machine has been derived from the semantics can be found at [14]. For the purpose of this paper, a short summary of the instructions follows.

Instruction *DECREGION* deletes from the heap all the regions, if any, between the current topmost region k and region k_0, excluding the latter. Each region can be deallocated with a time cost in $O(1)$ due to its implementation as a linked list (see [21] for details). Instruction *POPCONT* pops a continuation from the stack or stops the execution if there is none. Instruction *PUSHCONT* pushes a continuation. It will be used in the translation of a **let**.

Instructions *COPY* and *REUSE* just mimic the semantics given to the corresponding expressions. Instruction *CALL* jumps to a new instruction sequence and creates a new region. Function calls are always tail recursive, so there is no

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

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

Fig. 3. The *Safe* Virtual Machine (SVM)

need for a return instruction. Instruction MATCH does a jump depending on the constructor of the matched cell. The list of code labels $\overline{p_j}^m$ corresponds to the compilation of a set of **case** alternatives. Instruction $\mathit{MATCH!}$ additionally destroys the matched cell. The following invariant is ensured by the translation: For every instruction sequence in the code store cs, instruction i is the last one if and only if it belongs to the set $\{\mathit{POPCONT}, \mathit{CALL}, \mathit{MATCH}, \mathit{MATCH!}\}$.

Instruction $\mathit{BUILDENV}$ creates a portion of the 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 topmost region number k is pushed. Instruction $\mathit{BUILDCLS}$ allocates a fresh cell and fills it with a constructor application. It uses the same conventions as $\mathit{BUILDENV}$. Finally, instruction SLIDE removes some parts of the stack and it is used to remove environment fragments.

We have defined this semantics in Isabelle/HOL as the function:

$$\mathit{execSVM} :: \mathit{SafeImpProg} \Rightarrow \mathit{SVMState} \Rightarrow (\mathit{SVMState}, \mathit{SVMState})\ \mathit{Either}$$

where *Either* is a sum type and *SVMState* denotes a configuration $((h,k), k_0, pc, S)$ with the code store removed and the current instruction sequence replaced by a program counter $pc = (p,i)$. The code store cs is a read-only component and has been included in the type *SafeImpProg*. The current instruction can be retrieved by accessing the i-th element of the sequence $(cs\ p)$. If the result of $\mathit{execSVM}\ P\ s_1$ is $\mathit{Left}\ s_1$, this means that s_1 is a final state. Otherwise, it returns $\mathit{Right}\ s_2$.

3 The Translation

The translation splits the runtime environment (E_1, E_2) of the semantics into two: a compile-time one ρ mapping program variables to stack offsets, and the actual runtime environment contained in the stack. As this grows dynamically, numbers are assigned to the variables from the bottom of the environment. 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 of x.

An expression **let** $x_1 = e_1$ **in** e_2 will be translated by pushing to the stack a continuation for e_2, and then executing the translation of e_1. A continuation consists of a pair (k_0, p) where p points to the translation of e_2 and k_0 is the lower watermark associated to e_2. It is saved in the stack because the lower watermark of e_1 is different (see the semantics of *PUSHCONT*). As e_1 and e_2 share most of their runtime environments, the continuation is treated as a barrier below which the environment must not be deleted while e_2 has not reached its normal form. So, the whole compile-time environment ρ consists of a list of smaller environments $[\delta_1, \ldots, \delta_n]$, mimicking the stack layout. Each individual block i consists of a triple (δ_i, l_i, n_i) with an environment δ_i mapping variables to numbers in the range $(1 \ldots m_i)$, a block length $l_i = m_i + n_i$, and an indicator $n_i = 2$ 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:

1. $((\delta, m, 0) : \rho) + \{\overline{x_i \mapsto j_i}^{n}\} \stackrel{\text{def}}{=} (\delta \cup \{\overline{x_i \mapsto m + j_i}^{n}, m+n, 0) : \rho.$
2. $((\delta, m, 0) : \rho)^{++} \stackrel{\text{def}}{=} (\{\}, 0, 0) : (\delta, m+2, 2) : \rho.$
3. $\mathit{topDepth}\ ((\delta, m, 0) : \rho) \stackrel{\text{def}}{=} m$. 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 4 we show an idealised version of the translation function *trE* taking a *Core-Safe* expression and a compile-time environment, and giving as a result a list of SVM instructions and a code store. There, *NormalForm* ρ is the following list:

$$\mathit{NormalForm}\ \rho \stackrel{\text{def}}{=} [\mathit{SLIDE}\ 1\ (\mathit{topDepth}\ \rho), \mathit{DECREGION}, \mathit{POPCONT}]$$

The whole program translation is done by Isabelle/HOL function *trProg* which first translates each function definition by using function *trF*, and then the main expression by using *trE*. The source file is guaranteed to define a function before its use. The translation accumulates an environment *funm* mapping every function name to the initial bytecode sequence of its definition. The main part of *trProg* is:

```
$\mathit{trProg}\ (\mathit{datas}, \mathit{defs}, e) = ($
  **let** ...
    $((p, \mathit{funm}, \mathit{contm}), \mathit{codes}) = \mathit{mapAccumL}\ \mathit{trF}\ (1, \mathit{empty}, [\,])\ \mathit{defs};$
    $\mathit{cs} = \mathit{concat}\ \mathit{codes}$
  **in** ... $\mathit{cs}$ ...)
```

$$
\begin{array}{ll}
\mathit{trE}\ c\ \rho & = (\mathit{BUILDENV}\ [c] : \mathit{NormalForm}\ \rho,\ \{\}) \\
\mathit{trE}\ x\ \rho & = (\mathit{BUILDENV}\ [\rho\ x] : \mathit{NormalForm}\ \rho,\ \{\}) \\
\mathit{trE}\ (x@r)\ \rho & = (\mathit{BUILDENV}\ [\rho\ x, \rho\ r] : \mathit{COPY} : \mathit{NormalForm}\ \rho,\ \{\}) \\
\mathit{trE}\ (x!)\ \rho & = (\mathit{BUILDENV}\ [\rho\ x] : \mathit{REUSE} : \mathit{NormalForm}\ \rho,\ \{\}) \\
\mathit{trE}\ (a_1 \oplus a_2)\ \rho & = (\mathit{BUILDENV}\ [\rho\ a_1, \rho\ a_2] : \mathit{PRIMOP} : \mathit{NormalForm}\ \rho,\ \{\}) \\
\mathit{trE}\ (f\ \overline{a_i}^n\ @\ \overline{s_j}^m)\ \rho & = ([\mathit{BUILDENV}\ [\overline{\rho\ a_i}^n, \overline{\rho\ s_j}^m], \mathit{SLIDE}\ (n+m)\ (\mathit{topDepth}\ \rho), \mathit{CALL}\ p], \mathit{cs}') \\
\qquad \textbf{where} & (f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e) \in \mathit{defs} \\
 & \mathit{cs}' = \{p \mapsto \mathit{is}\} \cup \mathit{cs} \\
 & (\mathit{is}, \mathit{cs}) = \mathit{trE}\ e\ [(\{\overline{r_j \mapsto m - j + 1}^m, \overline{x_i \mapsto n - i + m + 1}^n\}, n+m, 0)] \\
\mathit{trE}\ (\textbf{let}\ x_1 = C_l^m\ \overline{a_i}^n @ s\ \textbf{in}\ e)\ \rho & = (\mathit{BUILDCLS}\ C_l^m\ [\overline{(\rho\ a_i)}^n]\ (\rho\ s) : \mathit{is},\ \mathit{cs}) \\
\qquad \textbf{where} & (\mathit{is},\ \mathit{cs}) = \mathit{trE}\ e\ (\rho + \{x_1 \mapsto 1\}) \\
\mathit{trE}\ (\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2)\ \rho & = (\mathit{PUSHCONT}\ p : \mathit{is}_1,\ \mathit{cs}_1 \cup \mathit{cs}_2 \cup \{p \mapsto \mathit{is}_2\}) \\
\qquad \textbf{where} & (\mathit{is}_1, \mathit{cs}_1) = \mathit{trE}\ e_1\ \rho^{++} \\
 & (\mathit{is}_2, \mathit{cs}_2) = \mathit{trE}\ e_2\ (\rho + \{x_1 \mapsto 1\}) \\
\mathit{trE}\ (\textbf{case}\ x\ \textbf{of}\ \overline{\mathit{alt}_i}^n)\ \rho & = ([\mathit{MATCH}\ (\rho\ x)\ \overline{p_i}^n],\ \{\overline{p_i \mapsto \mathit{is}_i}^n\} \cup (\bigcup_{i=1}^n \mathit{cs}_i)) \\
\qquad \textbf{where} & (\mathit{is}_i, \mathit{cs}_i) = \mathit{trA}\ \mathit{alt}_i\ \rho,\ \ 1 \le i \le n \\
\mathit{trE}\ (\textbf{case!}\ x\ \textbf{of}\ \overline{\mathit{alt}_i}^n)\ \rho & = ([\mathit{MATCH!}\ (\rho\ x)\ \overline{p_i}^n],\ \{\overline{p_i \mapsto \mathit{is}_i}^n\} \cup (\bigcup_{i=1}^n \mathit{cs}_i)) \\
\qquad \textbf{where} & (\mathit{is}_i, \mathit{cs}_i) = \mathit{trA}\ \mathit{alt}_i\ \rho,\ \ 1 \le i \le n \\
\mathit{trA}\ (C\ \overline{x_i}^n \to e)\ \rho & = \mathit{trE}\ e\ (\rho + \{\overline{x_i \mapsto n - i + 1}^n\})
\end{array}
$$

Fig. 4. Translation from *Core-Safe* expressions to *Safe-Imp* bytecode instructions

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

Fig. 5. Imperative code for the *Core-Safe* `appendD` program

where *cs* is the code store resulting from the compilation, and *mapAccumL* is a higher-order function, combining *map* and *foldl*, defined to Isabelle/HOL by copying its definition from the Haskell library (`http://dalila.sip.ucm.es/safe/theories` for more details).

In Figure 5 we show the code store generated for the following *Core-Safe* program with the `appendD` function of Sec. 2.1:

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

4 Formal Verification

The above infrastructure allows us to state and prove the main theorem expressing that the pair translation-abstract machine is sound and complete with respect to the resource-aware semantics. First, we make note that both the semantics and the SVM machine rules are syntax driven, and that their computations are deterministic (up to fresh names generation for the heap). So, we only need to prove that everything done by the semantics can be emulated by the machine, and that termination of the machine implies termination of the semantics (for the corresponding expression.)

First we define in Isabelle/HOL the following equivalence relation between runtime environments in the semantics and in the machine:

Definition 1. *We say that the environment $E = (E_1, E_2)$ and the pair (ρ, S) are equivalent, denoted $(E_1, E_2) \bowtie (\rho, S)$, if dom $E - \{self\} =$ dom ρ, and $\forall x \in$ dom $E_1 \,.\, E_1(x) = S!(\rho\ x)$, and $\forall r \in$ dom $E_2 - \{self\} \,.\, E_2(r) = S!(\rho\ r)$.*

Then we define an inductive relation expressing the evolution of the SVM machine up to some intermediate points corresponding to the end of the evaluation of sub-expressions:

```
inductive
  execSVMBalanced :: [SafeImpProg,SVMState,nat list,SVMState list,nat list] ⇒ bool
                     ( _ ⊢_ , _ -svm→ _ , _ )
where
  init: P ⊢ s, n#ns -svm→ [s], n#ns
| step: ⟦ P ⊢ s, n#ns -svm→ s'#ss, m#ms;
         execSVM P s' = Right s'';
         m' = nat (diffStack s'' s' m);
         m' ≥ 0;
         ms' = (if pushcont (instrSVM P s') then 0#m#ms
               else if popcont (instrSVM P s') ∧ ms=m''#ms'' then (Suc m'')#ms''
                    else m'#ms)⟧ ⟹
         P ⊢ s, n#ns -svm→ s''#s'#ss, ms'
```

$P \vdash s, n\#ns -svm\rightarrow ss, 1\#ns$ represents a 'balanced' execution of the SVM corresponding to the evaluation of a source expression. Its meaning is that the *Safe-Imp* program P evolves by starting at state s and passing through all the states in the list ss (s is the last state of the list ss, and the sequence progresses towards the head of the list), with the stack decreasing at most by n positions. Should the top instruction of the current state create a smaller stack, then the machine stops at that state. The symbol # in Isabelle/HOL is the *cons* constructor for lists.

Next, we define what resource consumption means at the machine level. Given a forwards state sequence $ss = s_0 \cdots s_r$ starting at s_0 with heap h_0 and stack S_0, *maxFreshCells ss* gives the highest non-negative difference in cells between the heaps in ss and the heap h_0. Likewise, *maxFreshWords ss* gives the maximum number of fresh words created in the stack during the sequence ss with respect to S_0. Finally, *diff k h h′* gives for each region j, $0 \leq j \leq k$, the signed difference in cells between h' and h.

From the input list *ds* of *Core-Safe* definitions, we define the set *definedFuns ds* of the function names defined there. Also, given an expression e, *closureCalled e ds* is an inductive set giving the names of the functions reached from e by direct or indirect invocation. By $cs \sqsubseteq cs'$ we mean that the code store cs' extends the code store cs with new bindings.

Finally, we show the correctness lemma of the semantics with respect to the machine, as it has been stated and proved in Isabelle/HOL:

```
lemma correctness:
  E ⊢ h , k , td , e ⇓ h' , k , v , r  ⟶
  (closureCalled e defs ⊆ definedFuns defs
  ∧ ((p, funm, contm), codes) = mapAccumL trF (1, empty, []) defs
  ∧ cs = concat codes
∧ P = ((cs, contm),p,ct,st)
∧ finite (dom h)
⟶ (∀ rho S S' k0 s0 p' q ls is is' cs1 j.
          (q, ls, is, cs1) = trE p' funm fname rho e
     ∧ (append cs1  [(q,is',fname)]) ⊑ cs
     ∧ drop j is' = is
     ∧  E ⋈ (rho,S)
     ∧ td = topDepth rho
     ∧ k0 ≤ k
     ∧ S' = drop td S
     ∧ s0 = ((h, k), k0, (q, j), S)
     ⟶ (∃ s ss q' i δ m w.
               P⊢s0 , td#tds -svm⟶ s # ss , 1#tds
        ∧ s = ((h', k) ↓ k0, k0, (q', i), Val v # S')
        ∧ fst (the (map_of cs q'))!i = POPCONT
        ∧ r = (δ,m,w)
        ∧ δ = diff k (h,k) (h',k)
        ∧ m = maxFreshCells (rev (s#ss))
        ∧ w = maxFreshWords (rev (s#ss)))))
```

The premises state that the arbitrary expression e is evaluated to a value v according to the *Core-Safe* semantics, that it is translated in the context of a closed *Core-Safe* program *defs* having a definition for every function reached from e, and that the instruction sequence *is* and the partial code store *cs1* are the result of the translation. Then, the execution of this sequence by the SVM starting at an appropriate state $s0$ in the context of the translated program P, will reach a stopping state s having the same heap (h', k) as the one obtained in the semantics, and the same value v on top of the stack. Moreover, the memory (δ, m, w) consumed by the machine, both in the heap and in the stack, is as predicted by the semantics.

The proof is done by induction on the $\Downarrow$ relation, and with the help of a number of auxiliary lemmas, some of them stating properties of the translation and some others stating properties of the evaluation. We classify them into the following groups:

Lemmas on the evolution of the SVM. This group takes care of the first three conclusions, i.e. *P⊢s0 , td#tds -svm⟶ s # ss , 1#tds* and the next two ones, and there is one or more lemmas for every syntactic form of e.

Lemmas on the equivalence of runtime environments. They are devoted to proving that the relation $(E_1, E_2) \bowtie (\rho, S)$ is preserved across evaluation. For instance, if $e \equiv f\ \overline{a_i}^n\ @\ \overline{r_j'}^l$, being f defined by the equation $f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e_f$, we prove that the equivalence of the environments local to f still hold. Formally:

$$
\begin{array}{ll}
 & (E_1, E_2) \bowtie (\rho, S) \\
\wedge & \rho' = [(\{\overline{x_i \mapsto n - i + l + 1}^n,\ \overline{r_j \mapsto l - j + 1}^l, \}, n + l, 0)] \\
\wedge & (E_1', E_2') = ([\overline{x_i \mapsto E(a_i)}^n], [\overline{r_j \mapsto E(r_j')}^l, \mathit{self} \mapsto k + 1]) \\
\wedge & S' = \overline{S!(\rho\ a_i)}^n\ @\ \overline{S!(\rho\ r_j')}^l\ @\ \mathit{drop\ td\ S} \\
\Longrightarrow & (E_1', E_2') \bowtie (\rho', S')
\end{array}
$$

Lemmas on cells charged to the heap. This group takes care of the last but two conclusion *δ = diff k (h,k) (h',k)*, and there is one or more lemmas for every syntactic form of e. For instance, if $e \equiv \textbf{let } x_1 = e_1 \textbf{ in } e_2$, then the main lemma has essentially this form:

$$\begin{array}{ll} & \delta_1 = \mathit{diff}\ k\ (h,k)\ (h'|_k,k) \\ \wedge & \delta_2 = \mathit{diff}\ k\ (h'|_k,k)\ (h'',k) \\ \Longrightarrow & \delta_1 + \delta_2 = \mathit{diff}\ k\ (h,k)\ (h'',k) \end{array}$$

where (h,k), $(h'|_k,k)$, and (h'',k) are respectively the initial heap, and the heaps after the evaluation of e_1 and e_2.

Lemmas on fresh cells needed in the heap. This group takes care of the last but one conclusion $m = \mathit{maxFreshCells}\ (\mathit{rev}\ (s\#ss))$. If $e \equiv \textbf{let } x_1 = e_1 \textbf{ in } e_2$, then the main lemma has essentially this form:

$$\begin{array}{ll} & \delta_1 = \mathit{diff}\ k\ (h,k)\ (h'|_k,k) \\ \wedge & m_1 = \mathit{maxFreshCells}\ (\mathit{rev}\ (s_1\#ss_1)) \\ \wedge & m_2 = \mathit{maxFreshCells}\ (\mathit{rev}\ (s_2\#ss_2)) \\ \Longrightarrow & \max\ m_1\ (m_2 + |\,\delta_1\,|) = \mathit{maxFreshCells}\ (\mathit{rev}\ (s_2\#ss_2\ @\ s_1\#ss_1\ @\ [s_0])) \end{array}$$

where s_0, s_1, and s_2 are respectively the initial state of the SVM, and the states after the evaluation of e_1 and e_2.

Lemmas on fresh words needed in the stack. This group takes care of the last conclusion $w = \mathit{maxFreshWords}\ (\mathit{rev}\ (s\#ss))$. If $e \equiv f\ \overline{a_i}^{n}\ @\ \overline{r'_j}^{l}$, then the main lemma has essentially this form:

$$\begin{array}{ll} & w = \mathit{maxFreshWords}\ (\mathit{rev}\ (s\#ss)) \\ \Longrightarrow & \max\ (n+l)\ (w+n+l-td) = \mathit{maxFreshWords}\ (\mathit{rev}\ (s\#ss\ @\ [s_2,s_1,s_0])) \end{array}$$

where s_0, s_1, s_2 are respectively the initial state of the application, and the states after the execution of *BUILDENV*and *SLIDE*, and $s\#ss$ is the state sequence of the body of f.

That termination of the SVM implies the existence of a derivation in the semantics for the corresponding expression has not been proved for the moment.

5 Discussion

On the use of Isabelle/HOL. The complete specification in Isabelle/HOL of the syntax and semantics of our languages, of the translation functions, the theorems and the proofs, represent almost one person-year of effort. Including comments, about 7000 lines of Isabelle/HOL scripts have been written, and about 200 lemmas proved.

Isabelle/HOL gives enough facilities for defining recursive and higher-order functions. These are written in much the same way as a programmer would do in ML or Haskell. We have not found special restrictions in this respect. The only 'difficulty' is that it is not possible to write potentially non-terminating functions. One must provide a termination proof when Isabelle/HOL cannot find one. Providing such a proof is not always easy because the argument depends on some other properties such as 'there are no cycles in the heap', which are not so easy to prove. Fortunately in these cases we have expressed the same ideas using inductive relations.

Isabelle/HOL also provides inductive n-relations, transitive closures as well as ordinary first-order logic. This has made it easy to express our 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 Higher-Order Logic has been essential for stating and proving the theorems.

Assessing how 'easy' it has been to conduct the proofs is another question. Part of the difficulties were related to our lack of experience in using Isabelle/HOL. The learning process was rather slow at the beginning. A second inconvenience is that proof assistants (as it must be) do not take anything for granted. Trivial facts that nobody cares to formalise in a hand-written proof, must be painfully stated and proved before they can be used. We have sparingly used the automatic proving commands such as `simp_all`, `auto`, etc., in part because they do 'too many' things, and frequently one does not recognise a lemma after using them. Also, we wanted the proof and to relate the proof to our hand-written version. As a consequence, it is very possible that our scripts are longer than needed. Finally, having programs and predicates 'living' together in a theorem has been an experience not always easy to deal with.

On the quality of the extracted code. The Haskell code extracted from the Isabelle/HOL definitions reaches 700 lines, and has undergone some changes before becoming operative in the compiler. One of these changes has been a trivial coercion between the Isabelle/HOL types *nat* and *int* and the Haskell type `Int`. The most important one has been the replacement of the Isabelle/HOL type $\rightharpoonup$ representing a partial function, heavily used for specifying our compile-time environments, by a highly trusty table type of the Haskell library. The code generated for $\rightharpoonup$ was just a λ-abstraction needing linear time in order to find the value associated to a key. This would lead to a quadratic compile time. Our table is implemented as a balanced tree and has also been used in other phases of the compiler. With this, the efficiency of the code generation phase is in $O(n \log n)$ for a single *Core-Safe* function of size n, and about linear with the number of functions of the input.

6 Related Work

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 [6]. 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 intensificaton at the end of the nineties. For instance, [19] uses a constraint solver to asses the validity of the GNU C compiler translations. They do not try to prove the compiler correct but instead to *validate its output* by comparing it with the corresponding input. This technique was originally proposed in [23]. A more recent experiment in compiler validation is [12]. In this case the source is the term language of HOL and the target is assembly language of the ARM processor. The compiler generates for each source, the object file and a proof showing that the semantics of the source is preserved. The last two stages of the compilation are in fact formally verified, while validation of the output is used in the previous phases.

More closely related to our work are [1] which certifies the translation of a Lisp subset to a stack language by using PVS, and [25] which uses Isabelle/HOL to formalise the translation from a small subset of Java (called μ-Java) to a stripped version of the Java Virtual Machine (17 bytecode instructions). Both specify the translation functions, and prove correctness theorems similar to ours. The latter work can be considered as a first attempt on Java, and it was considerably extended by Klein, Nipkow, Berghofer, and Strecker himself in [8,9,3]. Only [3] 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 [5,10,11]. 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 whith those of *gcc* running with level-2 optimisations activated. This is perhaps the biggest project on machine-assisted compiler verification done up to now.

Less related work are [7] and the MRG project [24], where certificates in Isabelle/HOL about heap consumption, based on special types inferred by the compiler, are produced. Two EU projects, EmBounded (`http://www.embounded.org`) and Mobius (`http://mobius.inria.fr`) have continued this work on certification and proof carrying code, the first one for the functional language Hume, and the second one for Java and the JVM.

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 shorter 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. We have improved on previous work by complementing functional correctness with a proof of resource consumption preservation.

References

1. Dold, A., Vialard, V.: A Mechanically Verified Compiling Specification for a Lisp Compiler. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245, pp. 144–155. Springer, Heidelberg (2001)
2. Barthe, G., Grégoire, B., Kunz, C., Rezk, T.: Certificate Translation for Optimizing Compilers. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 301–317. Springer, Heidelberg (2006)
3. Berghofer, S., Strecker, M.: Extracting a formally verified, fully executable compiler from a proof assistant. In: Proc. Compiler Optimization Meets Compiler Verification, COCV 2003. ENTCS, pp. 33–50 (2003)
4. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2004)
5. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 460–475. Springer, Heidelberg (2006)
6. Dave, M.A.: Compiler verification: a bibliography. SIGSOFT Software Engineering Notes 28(6), 2 (2003)
7. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: Proc. 30th ACM Symp. on Principles of Programming Languages, POPL 2003, pp. 185–197. ACM Press, New York (2003)

8. Klein, G., Nipkow, T.: Verified Bytecode Verifiers. Theoretical Computer Science 298, 583–626 (2003)
9. Klein, G., Nipkow, T.: 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)
10. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: Principles of Programming Languages, POPL 2006, pp. 42–54. ACM Press, New York (2006)
11. Leroy, X.: A formally verified compiler back-end, July 2008, p. 79 (submitted, 2008)
12. Li, G., Owens, S., Slind, K.: Structure of a Proof-Producing Compiler for a Subset of Higher Order Logic. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 205–219. Springer, Heidelberg (2007)
13. Lindholm, T., Yellin, F.: The Java Virtual Machine Sepecification, 2nd edn. The Java Series. Addison-Wesley, Reading (1999)
14. Montenegro, M., Peña, R., Segura, C.: A Resource-Aware Semantics and Abstract Machine for a Functional Language with Explicit Deallocation. In: Workshop on Functional and (Constraint) Logic Programming, WFLP 2008, Siena, Italy, July 2008, pp. 47–61 (2008) (to appear in ENTCS)
15. Montenegro, M., Peña, R., Segura, C.: A Simple Region Inference Algorithm for a First-Order Functional Language. In: Trends in Functional Programming, TFP 2008, Nijmegen (The Netherlands), May 2008, pp. 194–208 (2008)
16. Montenegro, M., Peña, R., Segura, C.: A Type System for Safe Memory Management and its Proof of Correctness. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 152–162. Springer, Heidelberg (1999)
17. Montenegro, M., Peña, R., Segura, C.: An Inference Algorithm for Guaranteeing Safe Destruction. In: LOPSTR 2008. LNCS, vol. 5438, pp. 135–151. Springer, Heidelberg (2009)
18. Necula, G.C.: Proof-Carrying Code. In: ACM SIGPLAN-SIGACT Principles of Programming Languages, POPL 1997, pp. 106–119. ACM Press, New York (1997)
19. Necula, G.C.: Translation validation for an optimizing compiler. SIGPLAN Notices 35(5), 83–94 (2000)
20. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL. A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
21. Peña, R., Rupérez, D.: A Certified Implementation of a Functional Virtual Machine on top of the Java Virtual Machine. In: Jornadas sobre Programación y Lenguajes, PROLE 2008, Gijón, Spain, October 2008, pp. 131–140 (2008)
22. Peña, R., Segura, C., Montenegro, M.: A Sharing Analysis for SAFE. In: Selected Papers of the 7th Symp. on Trends in Functional Programming, TFP 2006, pp. 109–128 (2007) (Intellect)
23. Pnueli, A., Siegel, M., Singerman, E.: Translation Validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
24. Sannela, D., Hofmann, M.: Mobile Resources Guarantees. EU Open FET project, IST 2001-33149 2001-2005, `http://www.dcs.ed.ac.uk/home/mrg`
25. Strecker, M.: Formal Verification of a Java Compiler in Isabelle. In: Voronkov, A. (ed.) CADE 2002. LNCS, vol. 2392, pp. 63–77. Springer, Heidelberg (2002)
26. Wildmoser, M.: Verified Proof Carrying Code. Ph.D. thesis, Institut für Informatik, Technical University Munchen (2005)