Programming a Smalltalk VM in Coq

Boris Shingarov
LabWare

Abstract

We describe an experimental attempt at verification of Smalltalk VM using mechanized proof. Only the native code generation part is verified. The generator is developed in the Coq proof assistant and is largely based on the CompCert compiler backend. The resulting VM successfully runs the ANSI test suite on a number of targets with different ISAs.

CCS Concepts • **Software and its engineering → Virtual machines**; **Formal software verification**;

Keywords Smalltalk Virtual Machine, certified compilation, program proof, Coq theorem prover, Curry–Howard correspondence

ACM Reference format:
Boris Shingarov. 2017. Programming a Smalltalk VM in Coq. In *Proceedings of IWST '17, Maribor, Slovenia, September 4–8, 2017,* 8 pages.
https://doi.org/10.1145/3139903.3139905

1 Introduction

The programming of critical systems, where human life or critical infrastructure is at stake, necessitates correctness assurances of a kind different than what we are used to in general-purpose programming. The *high assurance* of such systems is achieved via *formal proof*. In particular, the high-assurance community has formalized several programming language compilers and runtimes using automatic theorem provers.

Today, the reasoning methodologies used in the construction of practical Smalltalk virtual machines, remain completely informal. No machine-checked proof of a Smalltalk VM has ever been attempted. This renders Smalltalk unsuitable for programming high-assurance software.

In this paper, we describe "Verified Linear Smalltalk" — an experimental attempt to construct a proof of semantic preservation for a translation from Smalltalk bytecode to native machine code in the Coq proof assistant [4] using dependent types. The executable translator is automatically extracted from the proof.

For our experiment, we chose the Modtalk dialect of Smalltalk [9, 10] for two reasons:

- the low complexity of the bytecode set affords its formalization with a manageable amount of effort;
- the non-live nature of Modtalk's "Program Definition" allows easy reuse of parts of the CompCert AoT compiler; JIT compilation would have brought undue complexity.

Another conscious choice we made at the very beginning was compilation to machine code. It would be perhaps easier to construct a machine-checked Blue-Book-like Smalltalk interpreter in the style of Myreen and Gordon's Verified LISP [23]. However, no realistic Smalltalk implementation today can omit some sort of Deutsch–Schiffman-style native compilation [8]. Lest the result be "A Verified Toy Smalltalk", we could not avoid native machine code.

Section 2 of the paper rationalizes the boundaries of the part of Smalltalk we selected to formalize. In Section 3, we start with a representative example of how undefined behavior creeps into a Smalltalk VM, and use it to explain the gist of the idea how guarantees against such undefined behavior can be built with dependent types. Section 4 describes our experimental implementation. For illustration we give a few simple examples of proofs of theorems about tagged oops; these proofs are real Coq code taken from the actual VLS code. We explain how transformation from Modtalk bytecode to CompCert IR is programmed and proved. Section 5 discusses how the verifed (proven) and the unverified parts of our Smalltalk can be connected together without destroying the value of the proof. Section 6 presents the outcome of the experiment. Section 7 compares and relates our proof with other formalized language implementations. We conclude with an outlook towards future work in Section 8.

2 Scope of Verification

The Smalltalk system consists of many parts. Why concentrate on the correctness of the VM as the foundation for a trusted Smalltalk? The answer lies in the nature of proof as the basis of high-assurance software. Following the argument of Leroy [14], low-assurance software is validated by testing: the program is tried on a set of test vectors, and if it appears to behave as expected in each test, it is deemed acceptable. In this approach, bugs in core language implementation are a comparatively minor issue relative to the scope of the complete system. In contrast, the formal validation of high-assurance software does not rely on observation of

IWST '17, September 4–8, 2017, Maribor, Slovenia

ACM ISBN 978-1-4503-5554-4/17/09…$15.00
https://doi.org/10.1145/3139903.3139905

behavior. A downside of this approach is that now all levels of the software stack need to be equally strongly assured, because one "weak link" invalidates the proof of the whole. Until recently, compilers were such a weak link. For example, to aim understanding of C compiler bugs, Yang et al. built a test bench generating large sets of test vectors; their tool discovered 325 previously unknown cases of miscompilation in GCC and LLVM [29]. To address this problem, Leroy describes the formal certification of CompCert [14–18], a compiler for a C-like language. Running Yang's test on CompCert found zero bugs in the certified backend.

In our view, the problem is larger than just miscompilation (i.e. the compiler silently producing incorrect code). In more than one mainstream, industrial implementation of the Smalltalk VM, we have observed incorrect behaviors which by the logic of the VM's construction would be attributed to miscompilation by the underlying C compiler, but thorough consideration of the ISO C99 standard has shown that in fact no miscompilation was going on.[1] The common occurrence of this class of bugs in "real" Smalltalk VMs suggests the formalization of the VM as a first prerequisite before a larger-scope formalization can be attempted.

In the experiment being described here, we concentrate only on the native code generator part of the VM. (This is simply because we have to start somewhere; the formalization of other parts of the VM, such as the garbage collector, will be the topic of future research).

3 Proving Defined Behavior

Let's start with an example: consider some representative operation found in any Smalltalk VM, such as method lookup. Given object A and symbol S, we follow A's superclass chain to find the method with signature S; if we reach the end of the chain before finding such method, we effect the send of a "Message Not Understood" (MNU) message to A. In a typical implementation, we would be passed a pointer pA to the location of A in the "object heap"; at pre-agreed offsets at that location, there would be an "object header", a pointer to a "method dictionary array" (MDA), "first instance variable pointer", and so on. Our `lookup()` function needs to read the pointer to MDA from the specified offset from pA, dereference it, follow the MDA structure in search of symbol S, then possibly dereference the *superMDA* field, etc. The question to ask ourselves is: what are the guarantees that all these structures contain valid information, so that e.g. dereferencing a pointer does not result in a segmentation fault or other undefined behavior? How can I be sure that pA points to a valid location in the object heap? that $A.mda$ indeed points to an MDA? What if S doesn't point to a `Symbol` object?

In today's industrial VMs, it is not unusual to meet with assert code like this:

```
oop lookup(oop A, oop S)
{
  ...
  /* assert that we have a real MDA
   * because an unfound bug can destroy
   * contents of the MDA field
   */
  if (!is_mda_oop(A)) {
    Abort();
  }
  ...
}
```

Can I write a VM in a notation that allows to guarantee, with mathematical certainty, that such "unfound bugs" cannot happen at all?

Our answer to this question is based on notating the VM as a theorem in the formalism of Coq proof assistant. Coq approaches modularity from the point of view of "plug interfaces" where proof of the preconditions necessary to mechanically deduce the correctness of the module's execution, is passed in as an argument. To assert the validity of the proof is the same as to say the proof term has the correct type; i.e. that the actual parameter matches the formal parameter's type declaration.

To understand this idea more clearly, let's step away from `lookup()` momentarily and consider the trivial example of natural numbers. In Coq, $\mathbb{N}$ is defined as an inductive type,[2]

Inductive $\mathbb{N}$: Set :=
  $O : \mathbb{N}$ | $Succ : \mathbb{N} \rightarrow \mathbb{N}$.

It is easy to provide the usual inductive definitions of arithmetic operations such as "+", inductively prove properties such as commutativity of addition, etc. Consider the function *Pred*, which is the inverse of *Succ*. If the argument n is the successor of some m, then *Pred* returns m:

Definition Pred (n : $\mathbb{N}$) : $\mathbb{N}$:=
  match n with
  | $O \Rightarrow$????
  | $Succm \Rightarrow m$
  end.

But what if $n = O$? Zero has no predecessor.[3] If we consider the domain of *Pred* to be $\mathbb{N}$, then $Pred$ is a partial function. Its caller better not call it with $n = O$; but what if it does, *due to an unfound bug*? This is the line with "????" above:

[1] One striking example from this author's personal experience was when working on HPS — one of the most mature Smalltalk VMs — code for comparing IEEE floats suddenly started breaking after more than two decades of perfect stability. It turned out the C code in question was relying on reflexivity of "==" of floats. When $a == a$ evaluated to false, it looked like a C compiler bug, but close study of the ISO C standard reveals that this is valid bevahior — a nuance that evaded those VM engineers for decades.

[2] Note that the constructors O and $Succ$ do not have any "body".

[3] Different literature provides different definitions of $Pred$; some authors put $PredO = O$. We stick with our definition just for this illustration.

what shall we do in that case — call `Abort()`? The crucial idea is that we can make *Pred* into a total function by introducing a second argument of type "$n > 0$". When the caller wants to calculate the predecessor of 42, it calls *Pred* with 42 and a proof that $42 > 0$. The compiler performs this type-checking statically at compile time. If another caller does not know n ahead of time, it must nevertheless construct a proof to pass to *Pred* in a way that can be type-checked statically. A caller containing a path attempting to calculate $Pred(0)$ will refuse to compile.

This mechanism gives us a convenient notation in which the guarantees of well-defined behavior of our `lookup()` function can be naturally expressed. For example, we take the proof that *A.mda* points to an MDA as one of the arguments. `Lookup()`'s caller, `send()`, must pass this proof in. Accidentally, it happens to not construct it locally from zero hypotheses but to push this responsibility further up the call graph, but ultimately there is an inductive proof showing at which point the MDA was constructed, and at which point *A* was constructed and *A.mda* was filled with the pointer to the MDA. In Modtalk this is easier than in a "dynamic" Smalltalk, because the whole Program Definition is known ahead of time. In fact, trying to formalize `lookup()` leads one to appreciate some nuances of Modtalk's definition of MNU not present in other Smalltalk dialects. Of course in a language like Java, method lookup failure is a runtime error, whereas in Smalltalk this is a normal operation resulting in an MNU message. What if there is no method to handle MNU? There are three cases. You *can* have a valid program with no MNU handler. For example, `Nano` has no sends.[4] If the program does contain sends, they are late-bound, so there is no static guarantee against MNU. Trying to find a proof that the superclass chain terminates at a root, the proof assistant throws subgoals at the VM programmer discovering edge cases that violate VM integrity in interesting ways. For example, Smalltalk code can easily construct a cycle of superclasses, as "superclass" is just a regular instance variable in the class object; or equally easily wreak havoc in the metaobject protocol. Most VMs will indiscriminately allow such mutation and then crash when a message send is attempted. Modtalk's ObjectWriter may guarantee the consistency of the object memory's initial state, but we immediately see that e.g. the ability to dynamically update the superclass would break assurances about sends. At the time of writing, many of these subtleties are short-circuited in VLS by delegating them over the remote debug interface (see Section 5); only the actual generated n-code has been subject to rigorous proof.

[4]Nano is an example program coming with the Modtalk distribution. It is a "smallest" Modtalk program consisting only of main initializer returning 42. It has no classes, message sends, or object behavior.

4 Implementation

Our proof is implemented in Coq, delegating large portions of code generation to reused parts of Leroy's CompCert backend [14–18]. CompCert compiles to a wide range of target ISAs (IA32, AMD64, RISC/V, different variants of PowerPC, different variants of ARM) across various ABI flavours. CompCert is written in ML and Coq, and contains verified and unverified parts. Compilation consists of a number of passes. In the first (unverified ML) pass, source in the "Cminor" language is parsed down to an intermediate language "RTL". The verified compiler core (written in Coq and extracted into ML at CompCert's build time) performs a sequence of transformations from RTL through a number of intermediate languages: RTL ⟶ LTL ⟶ Linear ⟶ Mach ⟶ Target. This last language is an abstract representation of native machine code; an unverified Target Printer (written in ML) dumps it into native assembly text.

We plug into this infrastructure, as shown in Figure 1, by supplying a verified transformation pass from Smalltalk bytecode to Mach IR, programmed in Coq mimicking the pattern of other CompCert's passes.[5] Unlike some other frameworks for abstracting reasoning about instruction sets such as ArchC / ACCGen [3], CompCert does not have a unified view of "what a typical processor looks like". For example, CompCert's formalization of the IA32 ISA defines a separate, explicit abstraction for "addressing mode", while its PowerPC counterpart lacks such explicit abstraction. For this reason, we do not directly interface with late stages involved with the ISA.

4.1 Formalization of OOP tagging

We start by reusing Leroy's formalization of machine integer arithmetic.[6] Type "`int`" (machine unsigned) in CompCert is a record comprising a (mathematical) integer k (a member of $\mathbb{Z}$) and a proof that it fits in the machine word of size w (i.e. that $0 \leq k < m = 2^w$). Leroy proceeds to construct a bijection between these machine integers and $\mathbb{Z}$ mod m. He gives a Coq proof that this bijection is unique (in particular, not dependent on the exact *proof term* of $0 \leq k < m = 2^w$), followed by Coq proofs of a few basic properties, including properties of "machine integer arithmetic modulo m".

[5]In our experiments, we attempted both a transformation to Linear and a transformation to Mach, looking to reuse CompCert's infrastructure for dealing with ABI details when constructing Foreign Function Interfaces. For reasons explained in Section 8, this did not result in appreciable benefit. This failure was not critical for our experiment, because at the time of this writing VLS does not attempt FFI. For clarity, we are omitting Linear from Figure 1 and from discussion.

[6]In standard VM engineering practice today, reasoning about the word size, overflow, etc, is at best written in comments but more commonly is performed in the VM engineer's head. When we speak of formalization, we mean that the mathematical objects being operating upon by the programs we write, are proofs of properties of these machine integers and of the VM code manipulating them.

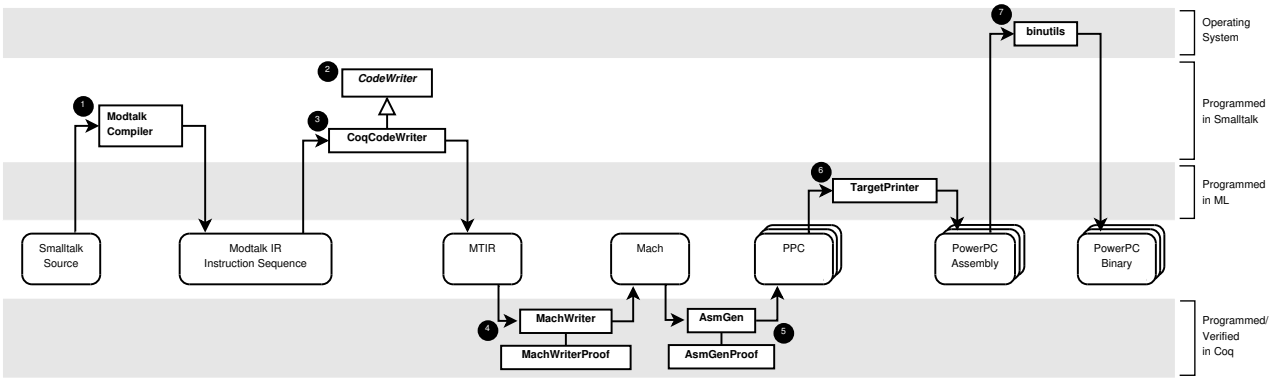

Figure 1. Relationship between Modtalk, VLS, Coq, and CompCert
In this figure, 1 and 2 are parts of Modtalk; 3 and 4 are parts of VLS; 5 and 6 are parts of CompCert, and 7 is an independent verified layer in the spirit of Piton [22] or part of the OS.

We build on this by defining a representation of SmallIntegers as a tagged machine word. Assume for simplicity that there is only one tag bit.[7] A SmallInteger consists of a number $x \in \mathbb{Z}$ and a proof that $0 \leq x < 2^{w-1}$:

```
Record SI: Type :=
  mkSI { intval: ℤ; intrange: -1 < intval < 2^(w-1) }.
```

How do we convert between SI and $\mathbb{Z}$? The integer meaning of a SmallInteger is given simply by projecting the first argument of the constructor:

```
Definition unsigned (n: SI) : ℤ := intval n.
```

In the opposite direction, instead of building a proof of $0 \leq x < 2^{w-1}$ every time we are given an $x \in \mathbb{Z}$, consider treating x, modulo sm (here $sm = 2^{w-1}$ is the "SmallInteger modulus"). The following lemma states that $0 \leq x \mod sm < sm$, and so $x \mod 2^{w-1}$ will always fit in a SmallInteger:[8]

```
Lemma M:
  ∀x, 0 ≤ Z_mod_modulus x < sm.
Proof.
  intros; unfold Z_mod_modulus.
  destruct x.
    - generalize small_modulus_pos; omega.
    - apply P_mod_two_p_range.
    - set (r := P_mod_two_p p (Int.wordsize - 1)).
      assert (0 ≤ r < sm) by apply P_mod_two_p_range.
      destruct (zeq r 0).
        + generalize small_modulus_pos; omega.
        + omega.
Qed.
```

The universal quantification "$\forall x$" makes this lemma a *dependent type*. The lemma's *statement* is a function taking an integer $x \in \mathbb{Z}$ and returning the proposition "$0 \leq x \mod sm < sm$" for that particular x, and the *proof* is a function constructively computing a *proof term* whose type is the above proposition. Now we pass this lemma as the second actual parameter to `mkSI`:

```
Definition repr (x : ℤ) : SI :=
  mkSI (Z_mod_modulus x) (Mx).
```

The machine *representation* is then given by the functions `int2si` and `si2int`, going from machine integers to SmallIntegers and back:[9]

```
Definition si2int (aSmallInteger: SI) : int :=
  Int.or
    (Int.shl (Int.repr (unsigned aSmallInteger)) (Int.repr 1))
    (Int.repr 1).

Definition int2si (aMachineInt: int) : SI :=
  repr (Int.unsigned (Int.shru aMachineInt (Int.repr 1))).

Example EncodeSmallInteger2:
     si2int (repr 2) = Int.repr 5.
Proof. reflexivity. Qed.

Example DecodeOop5:
     int2si (Int.repr 5) = repr 2.
Proof. reflexivity. Qed.
```

Note how `si2int` and `int2si` are defined simply in terms of machine integer arithmetic, whithout any regard to safety

[7] In actual Modtalk — and therefore VLS — there are more than one tag bits. Also, again for simplicity of illustration, let's ignore negative integers and their representation in two's complement.

[8] To understand the rest of the text, it is not necessary to understand the details of this proof which is based on Presburger arithmetic [24]. We are also omitting details of implementation of auxiliary functions such as Z_mod_modulus (which computes $a \mod b$).

[9] We are spelling out these definitions in full so that the reader can see what's going on. Coq's "Notation" facility affords to write such code rather elegantly, e.g. 1%Nat for the natural number 1, 1%Z for the integer 1, 1%int for the machine integer 1, 1%SmallInteger for the SmallInteger 1, 1%oop for SmallInteger zero's oop, etc. Further, the "Scope" facility allows to switch, depending on context, between which of these interpretations is meant by "1" written without a %-qualifier.

against wrapping around. In fact, the calls to `unsigned` function explicitly discard the witdh-fits guarantee passed in the argument. An alternative definition performs the bit manipulation on the side of $\mathbb{Z}$:

```
Definition int2si' (aMachineInt: int) : SI :=
  repr (Z.shiftr (Int.unsigned aMachineInt) 1).

Definition si2int' (aSmallInteger: SI) : int :=
  Int.repr ((Z.shiftl (unsigned aSmallInteger) 1) + 1).

Example EncodeSmallInteger2':
  si2int' (repr 2) = Int.repr 5.
Proof. reflexivity. Qed.

Example DecodeOop5':
  int2si' (Int.repr 5) = repr 2.
Proof. reflexivity. Qed.
```

Our last step before we turn to machine instruction sequences operating on tagged integers, is extending Leroy's definition of Value, which is "anything that can go into a machine word", to OOP which is any value that can be either a SmallInteger or a pointer OOP. We then describe the semantics of OOP arithmetic by defining functions such as

$$Oop.shru : int \rightarrow Oop \rightarrow Oop$$

as well as

$$Oop.toSmallInt : int \rightarrow Oop.$$

4.2 Tag Manipulation Code

Now we are ready to consider some canned code sequences for tag stripping and tag introduction, which we could then literally substitute into machine program text. The really interesting question is, from a proof perspective, how do we convince ourselves that these instruction sequences do the right thing? We use CompCert's "smart constructor" framework. The function `unary_constructor_sound` takes two arguments — an instruction constructor and a semantics over a value type — and returns a proposition stating that the instruction has the indicated semantics. We define the synthetic instruction `toSmallInt` as `shl 1` followed by `or 1` and then state theorem `eval_toSmallInt` by applying the function
(unary_constructor_sound toSmallInt Oop.toSmallInt).

4.3 The CoqCodeWriter

Figure 1 shows how the different components of VLS hold together. Like in any Smalltalk, Modtalk's frontend compiler transforms Smalltalk source code into CompiledMethods containing bytecoded Intermediate Representation.
Modtalk's backend is static in the sense that the whole program's IR is analyzed and transformed into executable code ahead of execution time. The backend is also pluggable in the sense that it is a framework of APIs which may be implemented by independent plugins. From this perspective, VLS is one such plugin. All backend implementations perform the same three stages of processing:

1. Object Allocation,
2. Code Writing, and
3. Object Initialization.

Stage 1, at which the statically-known memory structures are allocated and their references are stored into a global table, is implemented by the "XRef Writer" which is common to all backend plugins.

Stage 2, in which the actual code generation is performed, contains most of the dissimilarities between the different backends. Nevertheless, the general structure of this component, which is a Visitor traversing the IR sequences generated by the Smalltalk compiler, is defined in the abstract class `CodeWriter`. This class is part of the Modtalk framework, and each plugin must supply a concrete subclass. Typically, visiting each bytecode would bounce off the bytecode back into the code writer which has a separate method per bytecode class (i.e., `#visitPushR:`, `#visitReturn`, etc.) The class `CoqCodeWriter` does not implement such methods. It is on the unverified side of the verified/unverified interface, and as such, a major design goal was to make this part as trivial as possible. It is just an empty shell: all it does is dump the bytecode name and arguments to the stream which connects to the next transformation pass — the verified MTIR-to-Mach transformation.

At Stage 3, the "initialized data" segment is filled with known offsets and other such information. Its implementation is straightforward.

4.4 MTIR to Mach

There is only one transformation performed by VLS: from MTIR to Mach. This transformation is extremely simple: each MTIR instruction is replaced by a canned Mach instruction sequence. The nature of these sequences is not that much different from canned manually-selected instruction sequences familiar from many well-known Smalltalk JITs. The only thing Mach does for us is homogenizing out the idiosyncrasies of the concrete target instruction sets.

The really interesting question is then, how do we convince ourselves that the semantics of MTIR has been preserved? Leroy [15, pp.4–5, 16–18] explains various definitions of preservation (bisimulation, forward/backward simulation, preserving safety, etc.) This is critical for a C compiler because C programs are allowed multiple behaviors. In VLS, our life is much easier: Smalltalk bytecode is internally-deterministic. We are, therefore, interested in a general form of compiler's properties: if compiler $\mathfrak{C}$ compiled the source program S into the target program T:

$$\mathfrak{C} : S \rightarrow T$$

then we are interested in *some* relation $P(S, T)$.

There are, broadly, two opposite approaches to prove compiler correctness. On the one hand, a *verified* compiler is $\mathfrak{C}$

together with a proof (obtained by analysis of $\mathfrak{C}$ ahead of actual act of compilation) that

$$\forall S, (\mathfrak{C}(S) \downarrow T) \rightarrow P(S,T)$$

i.e. that if the compiler produces any code at all, that code will be correct (here, $\downarrow$ means "evaluates to").

On the other hand, a *certifying* compiler produces a target program T together with a proof of its correctness:

$$\forall S, \mathfrak{C}(S) \downarrow (T * P(S,T))$$

The crux of the difference between the two approaches is in the scope of "$\forall S$": in the former, it means "we prove once and for all input programs…"; in the latter, it means "every time we compile, we give a proof…"

The choice of approach is a matter of convenience (some transformations are easier to verify and some easier to certify) and the different compiler passes mix them freely.

Given what we already know in Subsection 4.2, constructing the verified MTIR transformation (approach 1) is straightforward. MTIR Instruction is an inductive type enumerating the known kinds of bytecodes, and the transformation works by case-analysis. Each case is a known instruction sequence just like we had in Subsection 4.2.

While this transformation seems trivial, the interesting aspect of it, is that we are forced to provide a formalization of what each bytecode does. This has not been attempted in Smalltalk before. Shingarov's target-agnostic code generator [27, 28] attempted to disentangle the definition of the bytecode from what the programmer believes the concrete processor instructions do, but what it manages to achieve is just to delay the problem by one move: the bytecodes are still *programmed* in what the programmer believes the meaning of the I/O effect specification to be.

5 Dealing with Incomplete Software

To write the code generator in one shot without trying out unfinished parts would be difficult; to write a whole VM in one project is impossible. There are, therefore, many parts of the VM that are outside the scope of our Verified Linear Smalltalk, such as primitives and memory management. There are many points of transfer of control between the verified and the unverified parts of the VM. How do we manage this complex relationship without destroying the value of the proof? In our view, Aarno and Engblom's virtual platform approach [1] provides a solution. In our experimental setup, the machine code generated by the verified code generator, runs under an inner/outer-Smalltalk debugger described in [26, 28]. No unverified part of the VM runs on the target. When an operation not implemented in the verified part needs to be performed, execution stop at a "surgery point" due to a processor trap. The Pharo host, which implements the client side of the GDB remote debug protocol, performs the necessary changes to the state of the target. This "surgery" is trusted by the target code: the Coq proof contains assumptions of the correctness of the state when control returns from the trap.

6 Results

The proof of a full VM such as those used in modern industrial Smalltalks, would exceed the scope of any reasonable research project. On the other hand, a VM for some "toy" subset of Smalltalk would be unconvincing, especially in light of the state of the art in other formally-verified programming languages. As a working compromise, we accepted the definition that a Smalltalk implementation is "realistic enough" if it runs the ANSI Smalltalk test suite. This coincides nicely with Modtalk's definition [9, 10].

Another "realness" criterion was that we generate code for real, not "toy", processors.

The resulting VM is able to run most of the ANSI test under the supervision of the inner-outer-Smalltalk debugger [28] on Intel and PowerPC ISAs both in GEM5 simulation [5] and on real hardware. We excluded "uninteresting" tests such as File I/O: even if we implemented them, it would be in the "outer Smalltalk supervisor" and this is not adding any new knowledge.

On the other hand, highly idiomatic Smalltalk constructs such as `#become:` would be interesting to implement and observe in a verified VM, but they are ignored by the X3J20 ANSI Standard, the ANSI test suite, and the Modtalk system.

The ANSI test, especially when executed within the SUnit framework, exhibits a lot of nuanced complexity, notably in its treatment of home contexts, block closures, and exceptions. Interestingly, (and somewhat contrary to expectation), running these tricky tests presented no nasty surprises. In Modtalk, the operational semantics of the tricky operations such as non-local return, are unambiguously defined in terms of straightforward manipulation of the virtual registers #A, #R, #X and fields in the activation record.

Our implementation cuts as many corners as possible without breaking ANSI. One such cut is the absence of hashing support in the VM: all hashes are taken to be 0. This, together with the total absence of any attempt at any sort of inline caching (a full lookup is performed on every send), results in so limited performance that any meaningful discussion of speed or speed measurement must be posponed until after at least some of these optimizations are implemented.

7 Related Work

Robinson–Voronkov [25] is the most complete reference on automated reasoning. Kaufmann et al. [13] is the standard monograph explaining modern automated reasoning technology in terms of a concrete system, ACL2, making it accessible for non-proof-theorists. Volume 1 [12] systematically treats the principles of automated reasoning and describes the ACL2 theorem prover in detail. Volume 2 [11] is devoted to practical applications of ACL2. Among them

are a formalization of IEEE754 floating-point multiplication used in validating the RTL design of floating-point circuits in AMD's microprocessors; a study of relationship between proved correct compiler and the Thompson attack; and numerous other applications.

Bertot–Castéran [4] is the standard reference on Coq.

To the best of our knowledge, to date the possibility of application of these or other reasoning systems to the Smalltalk VM has not been investigated.

Automatic reasoning has been used for automated verification of a number of programming language runtimes and compilers. We mention only those most relevant to our work:

Myreen and Gordon [23] provide a mechanical proof for a LISP interpreter in the HOL4 theorem prover.

Chlipala [7] verified a compiler from a functional language with references and exceptions to a toy assembly language.

Atkey et al.'s CoqJVM [2] and Liu and Moore's M6 [19] are both executable formal models of the Java VM. Neither of the two address Deutsch–Schiffman-like native execution. They are written in Coq and ACL2, respectively.

Historically the first formal proof of a compiler is [20]. The first mechanisation of such proof appeared in [21].

We use Leroy's CompCert [14–18] as the starting point for our experiment. CompCert is influential (216 citations according to ACM DL) and has a large community.

Shingarov's target-agnostic native code generator for Smalltalk [27, 28] uses logic programming to synthesize a Smalltalk VM from a formal definition of the ISA specified in a Processor Description Language (PDL), therefore completely relieving the human programmer from manual reasoning about the ISA. The PDL used there, is ArchC / ACCGen [3]. While the actual logical inference of the instruction sequences is performed using Prolog, the tool parses the ArchC specification into Smalltalk objects and allows many interesting kinds of reasoning about the ISA inside the Pharo environment.

Unlike Shingarov's target-agnostic tool, Verified Linear Smalltalk does not reason directly on the ISA, instead fully relying on CompCert for the final stages of machine code generation. This brings both some advantages and some disadvantages. For our purposes, CompCert has very satisfactory ISA support. CompCert's register allocation is more sophisticated than what can be achieved using the PDL approach. On the other hand, CompCert only goes up to assembly source, trusting the target's `binutils` for the assemble and link stages, whereas the PDL approach affords rich `binutils`-type functionality directly inside Pharo. This seems like a step back to the way Modtalk's *Native* backend performed these operations. Fully relying on CompCert also means losing a number of avenues for architectural analysis.

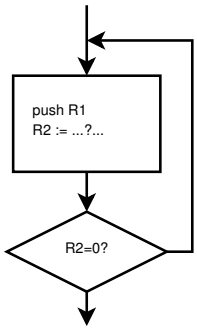

Figure 2. Whether a bytecode sequence pushes onto the stack infinitely, is undecidable

8 Future Work

The VLS experiment is currently a work in progress. At the time of this writing, the most severe limitations of VLS are caused by its explicit treatment of the stack, which in turn follows from the stack-oriented nature of our bytecode set. In order to take advantage of any part of the optimization frameworks present in modern compilers such as CompCert, there needs to be a separation between the stack-manipulation passes of compilation (e.g. register spilling, building of activation frame) and client program code which should be operating on pseudo-registers. Fundamental challenges to automatic reasoning about explicit stack manipulation include situations such as those illustrated in Figure 2, where the CompiledMethod is explicitly pushing onto the stack in a cycle. Another example is building the arguments of a Windows COM call from the function descriptor. Overcoming this challenge is especially interesting when combined with the prospect of selective inlining / Strongtalk-style specialization.

Our verified code generator uses many external technologies. At this stage of the experiment, the interoperation between host Pharo, Coq, ML, Menhir, CompCert etc. is extremely inefficient and confusing. While it would not make much sense to rewrite the trusted proof engine in Smalltalk, it would be interesting to integrate the ML implementation with Pharo in such a way that the state of the interactive proof session could be reflected upon in the GT Inspector [6], thus opening way to humane assessment of proofs.

We also intend to investigate what significant difference will it make to apply our approach to an image-based Smalltalk with dynamic compilation.

The technologies on which VLS is built, are themselves currently enjoying rapid advance. We hope to benefit from these new results in automatic theorem proving, compiler verification, and other relevant fields.

References

[1] D. Aarno and J. Engblom. 2015. *Software and System Development Using Virtual Platforms*. Elsevier.

[2] Robert Atkey. 2008. *CoqJVM: An Executable Specification of the Java Virtual Machine Using Dependent Types*. Springer Berlin Heidelberg, Berlin, Heidelberg, 18–32. https://doi.org/10.1007/978-3-540-68103-8_2

[3] R. Auler, P. C. Centoducatte, and E. Borin. 2012. ACCGen: An Automatic ArchC Compiler Generator. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*. 278–285.

[4] Y. Bertot and P. Castéran. 2004. *Interactive Theorem Proving and Program Development*. Springer, Berlin Heidelberg.

[5] N. L. Binkert et al. 2006. The M5 Simulator: Modeling Networked Systems. *IEEE Micro* 26, 4 (2006).

[6] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Gîrba. 2015. The Moldable Inspector. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. ACM, New York, NY, USA.

[7] Adam Chlipala. 2010. A Verified Compiler for an Impure Functional Language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA.

[8] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. ACM, New York, NY, USA, 297–302. https://doi.org/10.1145/800017.800542

[9] J. Fridstrom, A. Jacques, K. Kilpela, and J. Sarkela. 2014. Modtalk. In *8th Smalltalks — Argentina Conference*. Argentina, Córdoba.

[10] J. Fridstrom, A. Jacques, K. Kilpela, and J. Sarkela. 2015. Testing Modtalk. In *Lecture Notes in Business Information Processing, Agile Processes, in Software Engineering, and Extreme Programming — 16th International Conference, XP*. Helsinki, Finland.

[11] M. Kaufmann, P. Manolios, and J.S. Moore. 2000. *ACL2 Case Studies*. Volume 2 of *Advances in Formal Methods* [13].

[12] M. Kaufmann, P. Manolios, and J.S. Moore. 2000. *An Approach*. Volume 1 of *Advances in Formal Methods* [13].

[13] M. Kaufmann, P. Manolios, and J.S. Moore. 2000. Computer-Aided Reasoning. (2000).

[14] Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 42–54.

[15] Xavier Leroy. 2009. A formally verified compiler backend. In *Journal of Automated Reasoning*.

[16] Xavier Leroy. 2014. Compiler Verification for Fun and Profit. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design (FMCAD '14)*. FMCAD Inc, Austin, TX, Article 5, 1 pages. http://dl.acm.org/citation.cfm?id=2682923.2682930

[17] Xavier Leroy. 2017. *The CompCert C verified compiler*. INRIA, Paris.

[18] Xavier Leroy. July 2008. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2008), 107–115.

[19] H. Liu and J. S. Moore. 2003. Executable JVM Model for Analytical Reasoning: A Study. In *Workshop on Interpreters, Virtual Machines and Emulators*. ACM SIGPLAN, San Diego, California, USA.

[20] John McCarthy and James Painter. 1966. Correctness of a Compiler for Arithmetic Expressions. In *Mathematical Aspects of Computer Science (Proceedings of Symposia in Applied Mathematics)*, J.T Schwartz (Ed.), Vol. XIX. American Mathematical Society, 33–41.

[21] Robin Milner and Richard Weyhrauch. 1972. Proving compiler correctness in a mechanized logic. (1972).

[22] J Strother Moore. 1996. *Piton: a Mechanically Verified Assembly-Level Language*. Kluwer Academic Publishers.

[23] Magnus O. Myreen and Michael J. C. Gordon. 2009. Verified LISP Implementations on ARM, x86 and PowerPC. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 359–374.

[24] William Pugh. 1991. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, 4–13.

[25] Alan Robinson and Andrei Voronkov (Eds.). 2001. *Handbook of Automated Reasoning, volumes 1, 2*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands.

[26] B. Shingarov. 2014. Modern Problems for the Smalltalk VM. In *International Workshop on Smalltalk Technologies*. Cambridge, UK.

[27] B. Shingarov. 2014. The Synthesis of Target-Agnostic JIT. In *8th Smalltalks — Argentina Conference*. Argentina, Córdoba.

[28] B. Shingarov. 2015. Live Introspection of Target-Agnostic JIT in Simulation. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST '15)*. ACM, New York, NY, USA, Article 5, 9 pages. https://doi.org/10.1145/2811237.2811295

[29] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, 283–294.