Using Coq to generate and reason about x86 systems code

Nick Benton  Jonas Jensen  Andrew Kennedy
Microsoft Research  ITU Copenhagen  Microsoft Research

The talk describes some in-progress work on modelling, generating and reasoning about x86 code within the Coq proof assistant. This work is part of a larger project on compositional verification of high-level behavioural properties of low-level systems code, such as schedulers, memory managers, loaders and verifiers.

There are already a number of significant projects that involve verification of systems code, including work on the Hyper-V hypervisor and the seL4 kernel. One distinguishing feature of our approach is that we start with a clean slate, taking no dependencies on existing code, programming languages, or software architectures. Our trusted base consists only of (a) the hardware, (b) a model, in Coq, of a subset of the instruction set supported by the hardware, and, lest we forget, (c) Coq itself.

Our base is a formalization of a subset of (initially, purely sequential) x86 machine code. Since we wish to reason about the code that actually runs (including code that loads or otherwise produces executable code), this formalization is built on the binary representation of code in memory. Making good use of both dependency and the ssreflect libraries and extensions, we start with a library of definitions and lemmas concerning operations on n-bit words. Over this are defined machine configurations, instruction decoding and the transition semantics. We use an interesting monadic style to deal with the reading of subfields of instructions and the reading and mutation of components of configurations in the operational semantics.

A key novelty of our ‘minimal TCB’ approach is that we build everything inside Coq itself, relying on no external tools. Just how one should connect a conventional high-level language implementation with an interactive prover so as to support verification is a thorny issue, in terms of degree of assurance, expressivity and user interface. We are experimenting with cutting through this complexity by working entirely within the prover, which is, after all, the only single place where all the entities of interest (machine models, domain-specific models, programs, specifications and proofs) can represented and related. When typing

```
cocq MySystem.v
```

a binary image is produced, together with a proof that the machine-code and
Writing interesting software directly as raw machine code bits is, of course, rather tedious. But Coq provides plenty of facilities to make it more palatable, certainly for the size of program (up to the equivalent of a few thousand lines of C) we’re considering verifying interactively; one view of Coq is that it’s the world’s most powerful macro-assembler: the core type theory supports highly expressive abstractions, and its facilities for defining user notation, implicit coercions, and overloading, through type classes and canonical structures, support the definition of domain specific languages (and logics) with appealing surface syntax. A modest example of this is the embedding of x86 assembler syntax in Coq. The following example is valid in our Coq “.v” source file as well as in Microsoft’s assembler:

```assembly
mov ESI, heapInfo;
mov EDI, [ESI+0];
add EDI, bytes;
jc fail;
cmp [ESI+4], EDI;
jc fail;
mov [ESI+0], EDI
```

We have also used PHOAS to implement a convenient surface syntax for a small CPS-based functional language with a straightforward translation into machine code. Our hope is that what we lose by not using a conventional general purpose language will be partially offset, at least at the scale at which we are working, by the ability to use multiple domain-specific languages and code generators, in a setting where we can enforce strong guarantees on the way they interoperate.

In order to prove safety properties of machine code programs, we have developed a separation logic. The predicate $\text{safe } k \; P$ asserts that it is safe to execute $k$ steps from a machine state satisfying $P$. This is typically used in propositions such as

$$\forall k. \forall R. \text{safe } k \; (Q \ast R) \Rightarrow \text{safe } k \; (P \ast R).$$

For example, $P$ could describe a machine ready to perform a jump instruction, and $Q$ could describe a machine ready to run from the jump target. Any frame $R$ is preserved, and this holds for arbitrarily long-running programs, i.e., any $k$.

Since our specifications always preserve a frame and a step count, we define a specification logic over the basic separation logic assertions, following the work of Krishnaswami and of Birkedal et al. on higher-order frame rules, that captures this pattern, hiding the $k$ and $R$. A specification $S \in \text{spec}$ is a predicate on natural natural numbers and assertions, satisfying

$$\forall k, P, R. \forall k' \leq k. S \; k \; P \Rightarrow S \; k' \; (P \ast R).$$

1We provide a trivial, trusted, extractor that turns a Coq list ascii value into a binary file
Intuitively, a spec denotes how many steps the machine has to execute before it no longer holds and what frames the execution will preserve.

It turns out that safe ∈ spec and that (1) is equivalent to safe ⊗ Q ⊢ safe ⊗ P, using the tensor operator known from the literature on higher-order frame rules:

\[(S ⊗ R) \ k \ P := S \ k \ (P \ast R)\].

This lets us write compact and composable specifications where preservation of frames and counting of steps is implicit.

As in our previous work on proving compiler correctness for more idealized low-level machines, the specifications of code refer only to external behaviour rather than non-observable intermediate states, and the underlying semantics is based on ‘total’\(^\text{2}\), rather than partial, heaps (as memory management is done by code we verify, rather than baked into the machine model). We are currently working on broadening the notion of observation in the logic from simple termination to encompass properties of external interactions.

\(^{2}\text{Modulo there not being actual memory at every address, of course.}\)