Coq: The world's best macro assembler?

Andrew Kennedy Nick Benton
Microsoft Research
{akenn,nick}@microsoft.com

Jonas B. Jensen
ITU Copenhagen
jobr@itu.dk

Pierre-Evariste Dagand
University of Strathclyde
dagand@cis.strath.ac.uk

Abstract

We describe a Coq formalization of a subset of the x86 architecture. One emphasis of the model is brevity: using dependent types, type classes and notation we give the x86 semantics a makeover that counters its reputation for baroqueness. We model bits, bytes, and memory concretely using functions that can be computed inside Coq itself; concrete representations are mapped across to mathematical objects in the SSReflect library (naturals, and integers modulo 2^n) to prove theorems. Finally, we use notation to support conventional assembly code syntax inside Coq, including lexically-scoped labels. Ordinary Coq definitions serve as a powerful "macro" feature for everything from simple conditionals and loops to stack-allocated local variables and procedures with parameters. Assembly code can be assembled within Coq, producing a sequence of hex bytes. The assembler enjoys a correctness theorem relating machine code in memory to a separation-logic formula suitable for program verification.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theoray—Semantics; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Mechanical theorem proving

General Terms Languages

Keywords Coq, dependent types, assembly code

1. Introduction

The Coq proof assistant [27] is remarkably versatile, with applications that span deep, research-level mathematics [14], programming language metatheory [1] and compiler correctness for realistic languages [17]. As part of a larger project tackling verification of systems software, we have used Coq to *model* a subset of the x86 machine architecture, *generate* binary code for it, and *specify* and *prove* properties of that code [16]. This paper concerns the first two of these tasks, showcasing Coq as a rich language for giving very readable (and executable) semantics for instruction set architectures, and as a tool for both writing assembly language programs and generating machine code from them. Using the semantics and assembler as a foundation, the full power of Coq's abstraction and proof capabilities can be brought to bear on low-level programs.

PPDP '13, September 16 - 18 2013, Madrid, Spain Copyright is held by the owner/author(s). Publication rights licensed to ACM.

1.1 An example

Figure 1 presents code that computes the factorials of 10 and 12 and prints them to the console using an external `printf` function whose address is loaded from an indirection table. When we run

```
Definition call_cdecl3 f arg1 arg2 arg3 :=
  PUSH arg3;; PUSH arg2;; PUSH arg1;;
  CALL f;; ADD ESP, 12.

Definition main (printfSlot: DWORD) :=
  (* Argument in EBX *)
  letproc fact :=
    MOV EAX, 1;;
    MOV ECX, 1;;
      (* while ECX <= EBX *)
      while (CMP ECX, EBX) CC_LE true (
        MUL ECX;; (* Multiply EAX by ECX *)
        INC ECX
      )
  in
    LOCAL format;
      MOV EBX, 10;; callproc fact;;
      MOV EDI, printfSlot;;
      call_cdecl3 [EDI] format EBX EAX;;
      MOV EBX, 12;; callproc fact;;
      MOV EDI, printfSlot;;
      call_cdecl3 [EDI] format EBX EAX;;
      RET 0;;
    format:;;
      ds "Factorial of %d is %d";; db #10;; db #0.

Compute bytesToHex
  (assemble #x"C0000004" (main #x"C0000000")).
```

Figure 1. x86 factorial, assembled by Coq

`coqc` over this file, the code is assembled to produce a hexadecimal dump, which is then easily transformed to binary.

```
>coqc fact.v
     = "EB 1A B8 01 00 00 00 B9 01 00 00 00 EB 04 F7
   E1 FF C1 3B CB 0F 8E F4 FF FF FF FF E7 BB 0A 00 00
   00 BF 2C 00 00 C0 EB DA BF 00 00 00 C0 50 53 68 61
   00 00 C0 FF 17 81 C4 0C 00 00 00 BB 0C 00 00 00 BF
   4C 00 00 C0 EB BA BF 00 00 00 C0 50 53 68 61 00 00
   C0 FF 17 81 C4 0C 00 00 00 C3 46 61 63 74 6F 72 69
   61 6C 20 6F 66 20 25 64 20 69 73 20 25 64 0A 00 "
     : string
```

Even this tiny example shows the power of Coq as an assembler. Ordinary Coq definitions, such as `call_cdecl3`, serve as user-defined macros, here expanding to the standard calling sequence for a three-argument function that uses the x86 `cdecl` calling convention.

The `while` and `letproc` syntax are "built-in" macros that provide looping and procedural control constructs, hiding the use of scoped labels, branching, and a simple calling convention behind a useful abstraction. The `LOCAL` notation provides scoped labels, here used to identify inline data, created using `ds` and `db` directives for string and byte constants respectively.

Observe how assembly code syntax, familiar to users of tools such as MASM, is embedded directly in Coq source. Also note how the assembler itself is executed inside Coq (placing the code at fixed address `C0000004` and assuming that `printf`'s slot is at `C0000000`), and its output (a sequence of bytes) can be sent to the console in hexadecimal format.

One route to actually executing such code is to produce a standard executable format from within Coq, such as the Portable Executable (PE) form used by Microsoft Windows:

```
Definition bytes :=
  makePEfile #0 [::] EXE "winfact.exe" #x"00760000"
    [::Build_DLLImport "MSVCRT.DLL"
                       [::ImportByName "printf"]]
    (dd #0)
    (fun _ imports ⇒ main (hd #0 (hd nil imports))).

Compute bytesToHex bytes.
```

The `makePEfile` function requires a load address for the image, a list of imports from dynamically linked libraries (here, the `printf` function that is located in the C runtime library `MSVCRT.DLL`), and an assembler program, parameterized on the imports. Having run `coqc` over this file we can then run a trivial hex-to-binary tool to produce an executable:

```
>coqc winfact.v >winfact.hex
>hexbin winfact.hex winfact.exe
>winfact
Factorial of 10 is 3628800
Factorial of 12 is 479001600
```

Alternatively, we can run code on bare metal – or in a virtual machine – by appending a small boot loader, outside of Coq, and then constructing a CD image suitable for booting. Figure 2 shows a screen shot of a larger example created this way: the classic Game of Life written entirely in assembler in Coq.

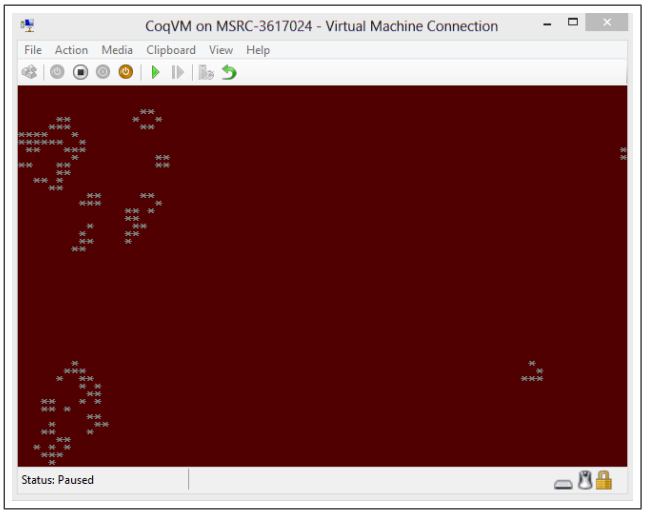

Figure 2. Game of Life

1.2 Contributions

Modelling. We show how Coq features such as dependent types, type classes, implicit coercions, indexed inductive types, and user-defined notation combine to support a clean formalization of machine code – in our case, a subset of x86. Many others have formalized machine code, but our approach has a number of interesting features:

- The representation of machine words and operations is concrete, using tuples of booleans to represent bit vectors, and implementing arithmetic and other operations directly as computable Coq functions, but types are informative, with fine-grained parameterization on word lengths. To prove theorems we exploit the embedding $\mathbb{B}^n \hookrightarrow \mathbb{N}$ of bit strings into the naturals and the isomorphism $\mathbb{B}^n \cong \mathbb{Z}_{2^n}$ of binary with with integers modulo 2^n, enabling use of a rich collection of lemmas from the SSReflect Mathematical Components library [15].
- An abstract, monadic treatment of *readers* and *writers*, with separate functional, imperative and logical interpretations, is unusual, and facilitates the reuse of binary format descriptions and proofs of *round-trip* properties.
- The semantics of execution itself is particularly concise, due mainly to our use of monadic syntax and careful factoring of the instruction type and auxiliary definitions.

Assembling. Our approach to generating code entirely inside Coq is novel:

- Coq notations and implicit coercions let us write assembly syntax that is cut-and-paste-compatible with standard Intel syntax.
- We support lexically-scoped labels within assembly code, through a kind of higher-order abstract syntax, and build verified abstractions such as control structures and procedure calling conventions over the top.
- We have proved that our instruction decoder is a left-inverse of our instruction encoder and that our assembler preserves correctness proofs of the assembled programs.
- We show how a verified compiler for regular expressions may be constructed, building on a third-party formalization of the theory of Kleene algebras.

2. Modelling x86

The x86 architecture and instruction set is notoriously complex: the current Intel manual detailing the instruction set alone runs to 1288 pages [11]. Although we so far model only a small subset, even here we must get to grips with rich addressing modes and a variable-length, non-uniform instruction format. In addition to aiming for *brevity* in our Coq description – to facilitate readability and extension – we also wanted an executable semantics. Running the first two instructions from the example in Figure 1 produces the following state changes (assuming that the initial value of `ESP` is `D0000000`):

```
Coq < Compute procStateToString (runFor 0 s).
 = " EIP=C0000000 ESP=D0000000 EBP=00000000
     EAX=00000000 EBX=00000000 ..."
Coq < Compute procStateToString (runFor 1 s).
 = " EIP=C0000001 ESP=CFFFFFFC EBP=00000000
     EAX=00000000 EBX=00000000 ..."
Coq < Compute procStateToString (runFor 2 s).
 = " EIP=C0000003 ESP=CFFFFFFC EBP=CFFFFFFC
     EAX=00000000 EBX=00000000 ..."
```

2.1 Bits, bytes and all that

We start with a concrete representation for bit vectors.

```
Definition BITS n := n.-tuple bool.
Definition NIBBLE := BITS 4.
```

```
Definition BYTE := BITS 8.
Definition WORD := BITS 16.
Definition DWORD := BITS 32.
Definition QWORD := BITS 64.
Definition DWORDorBYTE (d: bool) :=
  BITS (if d then 32 else 8).
```

The `n.-tuple` type of SSReflect is used to represent n-bit words concretely and efficiently, with synonyms defined for bytes, 16-bit words and 32-bit words. The `DWORDorBYTE` type is used for literals in instructions that have byte and 32-bit word variants, and illustrates well how 'lightweight' dependent types are handy for formalizing machine semantics. Syntax is provided for decimal, hexadecimal, binary and character constants:

```
Example fortytwo := #42 : BYTE.
Example fortytwo1 := #x"2A".
Example fortytwo2 := #b"00101010".
Example fortytwo3 := #c"*".
```

The notations `#x` and `#b` expand to the following two functions, demonstrating another application of dependent types:

```
Fixpoint fromHex s : BITS (length s * 4) :=
  if s is String c s
  then joinNibble (charToNibble c) (fromHex s) else #0.


Fixpoint fromBin s : BITS (length s) :=
  if s is String c s
  then joinmsb (charToBit c, fromBin s) else #0.
```

The `if ... is` notation is SSReflect syntactic sugar for `match`.

Using indexing in the type of words allows accurate typing of machine operations such as arithmetic, shifts, and rotates. For example, the "carry-out" of operations such as addition and shift-left is simply expressed as the most significant bit of the result, and full bit-length multiplication produces a result whose size is the sum of the sizes of its inputs:

```
Fixpoint adcB n carry : BITS n → BITS n → BITS n.+1.
Definition dropmsb {n} : BITS n.+1 → BITS n.
Definition addB {n} (p1 p2: BITS n) :=
                      dropmsb (adcB false p1 p2).
Definition fullmulB {n1 n2} : BITS n1 → BITS n2
                           → BITS (n1+n2).
Definition shlB {n} : BITS n → BITS n.+1.
Definition catB {n1 n2} : BITS n1 → BITS n2
                        → BITS (n2+n1).
Notation "x ## y" := (catB x y)
             (right associativity, at level 60).
```

This concrete representation also makes computation over words relatively efficient; this is important as both code generation and the direct execution of the machine model for testing purposes involve performing substantial bit-level computation inside Coq. Performance is adequate for these purposes: a quick test reveals that we can achieve roughly 150,000 32-bit increments per second, and 11,000 32-bit additions (on a 3.6Ghz Intel Xeon CPU).

Proofs of properties of word operations often proceed by induction on n. For arithmetic, however, we found it useful to map machine representations onto more abstract mathematical types provided by SSReflect, either embedding `BITS n` into `nat` or using the bijection with `'Z_(2^n)`, the type of integers modulo 2^n. (Note that for both of these forms, representations are unique, so we can reason using standard Coq equality.) Here, for example, is the proof that addition is associative:

```
Lemma addBA n : associative (@addB n).
Proof. move ⇒ x y z. destruct n; first apply trivialBits.
  apply toZp_inj. autorewrite with ZpHom.
  by rewrite addrA. Qed.
```

The first line of the proof discharges the trivial case of zero bits. Next we use the lemma `toZp_inj`, which states that

```
toZp: BITS n → 'Z_(2^n)
```

is injective, in order to transform the goal from

```
  addB x (addB y z) = addB (addB x y) z
```

to

```
toZp (addB x (addB y z)) = toZp (addB (x y) z).
```

We then push the embedding inwards by appealing to the hint database `ZpHom`, which contains lemmas expressing that `toZp` behaves homomorphically with respect to all relevant arithmetic operations – in particular `addB` and addition on the ring. Finally we use the associativity of addition on the target, as provided by the SSReflect library. Many other properties are proved using the same recipe.

2.2 Memory

To model memory we use an implementation of finite partial maps whose domain is n-bit words. We could define this abstractly using `BITS n -> option V`, but as with machine words we prefer something more concrete that can be used for efficient execution of instruction semantics inside Coq. We use a variant of *tries*, splitting on a bit of the address at each level, with `NEPMAP n` representing non-empty maps, and `PMAP n` for possibly-empty maps:

```
Variable V: Type.
Inductive NEPMAP : nat → Type :=
| VAL   : V → NEPMAP 0
| SPLIT : ∀ n (lo hi: NEPMAP n), NEPMAP n.+1
| LSPLIT : ∀ n (lo : NEPMAP n), NEPMAP n.+1
| RSPLIT : ∀ n (hi : NEPMAP n), NEPMAP n.+1.
Inductive PMAP n :=
| PMap     : NEPMAP n → PMAP n
| EmptyPMap : PMAP n.


Definition lookup n (m: PMAP n) (p: BITS n) : option V
 := if m is PMap m' then lookupNE m' p else None.
Global Coercion lookup : PMAP ⟼ Funclass.
```

where `lookupNE` recurses down an `NEPMAP`; at each non-leaf level, the constructor determines whether the left, right, or both subtrees are present. A benefit of this fussiness is uniqueness of representation: two values of type `PMAP n` are equal in Coq if and only if they are extensionally equal. This is expressed by the following lemma, which makes use of the implicit coercion between function application and `lookup`:

```
Lemma extensional_PMAP n V:
  ∀ (m1 m2: PMAP V n), (∀  x, m1 x = m2 x) → m1 = m2.
```

Memory is then a partial map from 32-bit addresses to bytes, in which the absence of an element indicates that the memory is not mapped, or is inaccessible:

```
Definition Mem := PMAP BYTE 32.
```

In future we plan to refine the model to account for non-writable and non-executable memory.

2.3 Monads

To abstract a little from the details of execution, decoding, assembling, and so on, we make use of monads. We employ Coq's type classes for packaging [25], defining a class `MonadOps` for syntax and `Monad` for the monad laws, along with some useful notation.

```
Class MonadOps T :=
{ retn {X} : X → T X
; bind {X Y} : T X → (X → T Y) → T Y }.
```

```
Class Monad T {ops: MonadOps T} :=
{ id_l X Y (x: X) (f: X → T Y) : bind (retn x) f = f x
; id_r X (c: T X) : bind c retn = c
; assoc X Y Z (c: T X) (f: X → T Y) (g : Y → T Z) :
  bind (bind c f) g = bind c (fun x ⇒ bind (f x) g) }.

Notation "'let!' x = c ; d" := (bind c (fun x ⇒ d)) (...)
Notation "'do!' c ; d" := (bind c (fun _ ⇒ d))
```

Concrete instances of `Monad` include an error monad and state monad transformer, both of which are used in the semantics of instruction execution. For reading and writing memory, and decoding and encoding of instructions, we define reader and writer monads, which are discussed next.

2.4 Readers and writers

Reading and writing sequences of bytes representing some other type of data, such as 32-bit words, or variable-length instructions, pervades our framework. We found it surprisingly difficult to devise an appropriate interface for these operations.

Firstly, there are issues with edge cases. Typically one has a pointer which is advanced as bytes are read or written; but the situation at the top-of-memory must be handled, and we wish to model contiguous memory ranges, possibly empty, and possibly all of memory. To tackle this we introduced a `Cursor` type, a value of which is either a concrete n-bit address, or a value `top` representing the address just beyond the end of memory.

```
Variable n:nat.
Inductive Cursor := mkCursor (p: BITS n) | top.
```

When reading or writing bytes, the cursor is advanced, but does *not* wrap around to zero, instead taking on the value `top`. (Compare the notion of EOF used in file I/O.) Memory ranges can be represented by a pair of cursors (p,q) interpreted as the memory from p inclusive to q exclusive.

A second issue was the need for multiple 'views' of reading and writing. Sometimes we wish to view reading and writing in a pure, functional style, in which a reader consumes, and a writer produces, a sequence of bytes. In other situations we want to make reads and writes on our concrete model of memory, for example, when specifying the execution behaviour of instructions. A third view, used in specifications, interprets readers and writers as *predicates* on partial states, with sequencing interpreted by separating conjunction.

To support these various views of reading and writing, we introduce inductively-defined *terms* for readers and writers.

Readers. A reader is defined as follows:

```
Inductive ReaderTm T :=
| readerRetn (x: T)
| readerNext (rd: BYTE → ReaderTm T)
| readerCursor (rd: Cursor 32 → ReaderTm T).

Class Reader T := getReaderTm : ReaderTm T.
Instance readBYTE : Reader BYTE :=
  readerNext (fun b ⇒ readerRetn b).
```

A reader for type `T` either returns a value of type `T` immediately, or consumes a single byte and then continues, or asks for the current value of the cursor and continues. This last feature is used in instruction decoding to implement relative addressing for branch instructions. Given appropriate definitions for monad unit and bind operations, we can define `MonadOps Reader` and `Monad Reader` instances, and easily create readers for various types. For example, here is a reader for 32-bit words, in little-endian format:

```
Definition bytesToDWORD (b3 b2 b1 b0: BYTE) : DWORD :=
  b3 ## b2 ## b1 ## b0.
Instance readDWORD : Reader DWORD :=
  let! b0 = readNext;
  let! b1 = readNext;
  let! b2 = readNext;
  let! b3 = readNext;
  retn (bytesToDWORD b3 b2 b1 b0).
```

Let's now interpret readers. First, functionally:

```
Definition runReader T :
  Reader T → Cursor 32 → seq BYTE →
               option (Cursor 32 * seq BYTE * T).
```

Given a list of bytes and an initial cursor position, bytes are read sequentially, resulting in a new position (possibly `top`), residual bytes, and a value; the value `None` is returned if there are insufficient bytes or if bytes at `top` are read.

Our second interpretation is imperative and operates on our model of memory, `Mem`. This time we distinguish between reading beyond the end of memory (`readerWrap`), and reading an unmapped byte (`readerFail`).

```
Inductive readerResult T :=
  readerOk (x: T) (q: Cursor 32)
| readerWrap | readerFail.
Definition readMem T :
  Reader T → Cursor 32 → Mem → readerResult T.
```

Finally, we can give a *logical* view of a reader, expressed as a 'store predicate' (`SPred`) which we discuss in detail elsewhere [16].

```
Definition interpReader T:
  Reader T → Cursor 32 → Cursor 32 → T → SPred.
```

Given a reader `R` for type `T`, the predicate `interpReader R p q x` holds of a state if the bytes between cursors `p` and `q` can be 'read back' as value `x` of type `T`.

Writers. For writers, we again define a term syntax with appropriate monadic unit and bind operations; a writer for type `T` is then a function from `T` to `WriterTm unit`.

```
Inductive WriterTm A :=
| writerRetn (x: A)
| writerNext (b: BYTE) (w: WriterTm A)
| writerCursor (w: Cursor 32 → WriterTm A)
| writerFail.
Class Writer T := getWriterTm: T → WriterTm unit.
```

As with readers, our monadic syntax makes the definition of writers for various types straightforward:

```
Instance writeBYTE : Writer BYTE :=
  fun b ⇒ writerNext b (writerRetn tt).
Instance writeDWORD : Writer DWORD := fun d ⇒
  let: (b3,b2,b1,b0) := split4 8 8 8 8 d in
  do! writeBYTE b0;
  do! writeBYTE b1;
  do! writeBYTE b2;
  do! writeBYTE b3;
  retn tt.
```

Writers have functional, imperative, and logical interpretations:

```
Definition runWriter T :
  Writer T → Cursor 32 → T → option (seq BYTE).
Definition writeMem T :
  Writer T → Cursor 32 → T → Mem →
    option (Cursor 32 * Mem).
Definition interpWriter T :
  Writer T → Cursor 32 → Cursor 32 → T → SPred.
```

To show that a reader correctly decodes anything produced by a writer, we construct an inductively defined *simulation relation* between them. The relation `simrw x p R W` says that reader `R` simulates writer `W` for the purpose of reading value `x` starting at position

p, meaning that R and W essentially proceed in lock step, except that they are allowed to read the cursor position independently of each other. If the writer fails, there is no restriction on the reader.

```
Inductive simrw {X T} (x: X) :
  Cursor 32 → Reader X → WriterTm T → Prop :=
| simrw_retn p t:
    simrw x p (readerRetn x) (writerRetn t)
| simrw_next p R b W':
    simrw x (next p) (R b) W' →
    simrw x p (readerNext R) (writerNext b W')
| simrw_rcursor p R' W:
    simrw x p (R' p) W →
    simrw x p (readerCursor R') W
| simrw_wcursor p R W':
    simrw x p R (W' p) →
    simrw x p R (writerCursor W')
| simrw_fail p R:
    simrw x p R writerFail
| simrw_top R b W':
    simrw x (top _) R (writerNext b W').
```

If for any x and position p a reader R and writer W are related by simrw then they satisfy a *round-trip* property:

```
Class Roundtrip X (R: Reader X) (W: Writer X) :=
  roundtrip: ∀ x p, simrw x p R (W x).
```

The following implication about round-tripping holds in our program logic [16], which intuitively means "if memory from p to q contains x as written by W, then reading back that range of memory with R produces the same x".

```
Lemma interpWriter_roundtrip X (W: Writer X) (R: Reader X)
      {RT: Roundtrip R W} p q x:
  interpWriter p q x ⊢ interpReader p q x.
```

The X, R and W arguments to the interpretations above are implicit.

2.5 Instructions

The x86 instruction set design is complex, and shows its history. However, some structure can be discerned, and is sufficient to support the definition of a Coq inductive type for instructions that is *total*: every value in the type represents a valid instruction, for which there is a defined encoding.

Registers. The 32-bit x86 processor has eight general-purpose registers, a flags register (EFLAGS), and an instruction pointer (EIP). (We do not yet model the legacy segment registers, FPU registers or SIMD registers.) We further divide the general-purpose registers into ESP (the stack pointer) and the remainder, because ESP is not allowed to participate in certain addressing modes.

```
Inductive NonSPReg :=
| EAX | EBX | ECX | EDX | ESI | EDI | EBP.
Inductive Reg := nonSPReg (r: NonSPReg) | ESP.
Inductive AnyReg := regToAnyReg (r: Reg) | EIP.
```

Some instructions can address the original 8-bit subregisters of the 8086.

```
Inductive BYTEReg := AL|BL|CL|DL|AH|BH|CH|DH.
Definition DWORDorBYTEReg (d: bool) :=
  if d then Reg else BYTEReg.
```

Addressing modes. The addressing modes used by most instructions are captured by the inductive definitions of Figure 3. To take an example, binary operations such as arithmetic and logical operations take two operands, one of which is also used as the destination. The instruction `add EAX, [EBX + ECX*4 + 14]` makes use of the most complex addressing mode, indirecting through an address that is computed as the sum of a base register EBX, an index register ECX scaled by 4, and the fixed offset 14. This would be represented by

```
Inductive Scale := S1 | S2 | S4 | S8.
Inductive MemSpec :=
  mkMemSpec (base: Reg)
            (indexAndScale: option (NonSPReg*Scale))
            (offset: DWORD).
Inductive RegMem d :=
| RegMemR (r: DWORDorBYTEReg d)
| RegMemM (ms: MemSpec).
Inductive Src :=
| SrcI (c: DWORD)
| SrcM (ms: MemSpec)
| SrcR (r: Reg).
Inductive DstSrc (d: bool) :=
| DstSrcRR (dst src: DWORDorBYTEReg d)
| DstSrcRM (dst: DWORDorBYTEReg d) (src: MemSpec)
| DstSrcMR (dst: MemSpec) (src: DWORDorBYTEReg d)
| DstSrcRI (dst: DWORDorBYTEReg d) (c: DWORDorBYTE d)
| DstSrcMI (dst: MemSpec) (c: DWORDorBYTE d).
```

Figure 3. Addressing modes

```
Inductive BinOp :=
| OP_ADC | OP_ADD | OP_AND | OP_CMP
| OP_OR | OP_SBB | OP_SUB | OP_XOR.
Inductive UnaryOp :=
| OP_INC | OP_DEC | OP_NOT | OP_NEG.
Inductive Condition :=
| CC_O | CC_B | CC_Z | CC_BE | CC_S | CC_P | CC_L | CC_LE.
Inductive Instr :=
| UOP d (op: UnaryOp) (dst: RegMem d)
| BOP d (op: BinOp ) (ds: DstSrc d)
| BITOP (op: BitOp) (dst: RegMem true) (bit: RegImm false)
| TESTOP d (dst: RegMem d) (src: RegImm d)
| MOVOP d (ds: DstSrc d)
| MOVX (signextend w:bool) (dst: Reg) (src: RegMem w)
| SHIFTOP d (op: ShiftOp) (dst: RegMem d) (count: ShiftCount)
| MUL {d} (src: RegMem d)
| IMUL (dst: Reg) (src: RegMem true)
| LEA (reg: Reg) (src: RegMem true)
| JCC (cc: Condition) (cv: bool) (tgt: Tgt)
| PUSH (src: Src)
| POP (dst: RegMem true)
| CALL (tgt: JmpTgt) | JMP (tgt: JmpTgt)
| CLC | STC | CMC | NOP
| RETOP (size: WORD)
| OUT (d: bool) (port: BYTE)
| IN (d: bool) (port: BYTE)
| HLT | BADINSTR.
```

Figure 4. Supported instruction set

```
DstSrcRM EAX (mkMemSpec EBX (Some(ECX, S4)) #14).
```

Instructions. Now let's tackle the instructions themselves. We abstract just far enough above the binary encoding, but not too far. Sometimes there is more than one way to encode essentially the same instruction, and we do not make this distinction in the data type. Furthermore, the encoding of branches and certain calls uses EIP-relative addressing. Our data type uses absolute addresses uniformly – this makes it slightly easier to formalize the execution semantics, and *much* easier to support scoped labels and proof rules for control-flow instructions, as code addresses are uniform. The instructions we have currently formalized are shown in Figure 4. Notice the extensive use of dependency (the d parameter) in instructions such as MOV that have two flavours, one for bytes, the other for 32-bit words.

2.6 Operational semantics

In essence, the machine is modelled as a state-to-state transition function, a single step consisting of decoding the next instruction and then executing it. The function is partial – because we only model a subset of instructions, and some behaviour is left unspecified – and so we structure it monadically, layering an `option` monad over a state monad.

Machine state. The machine state splits into three: registers, flags, and memory. Register state is modelled as a (finite) function:

```
Definition RegState := AnyReg → DWORD.
```

We model flags separately, in order to model the unspecified effect that some x86 instructions have on them.

```
Definition Flag := BITS 5.
Definition CF: Flag := #0. Definition PF: Flag := #2.
Definition ZF: Flag := #6. Definition SF: Flag := #7.
Definition OF: Flag := #11.
Inductive FlagVal := mkFlag (b: bool) | FlagUnspecified.
Coercion mkFlag : bool >-> FlagVal.
Definition FlagState := Flag → FlagVal.
```

Putting registers, flags, and memory together gives us the processor state:

```
Record ProcState := mkProcState
{ registers:> RegState; flags:> FlagState; memory:> Mem }.
```

Instruction decoding. Instruction decoding is implemented as an instance of `Reader`, making good use of the monadic syntax mixed with simple Coq computation. The fragment here, for example, has just decoded an `FE` byte:

```
let! (opx,dst) = readNext;
if opx == #b"000" then
  retn (UOP false OP_INC dst)
else
if opx == #b"001" then
  retn (UOP false OP_DEC dst)
else retn BADINSTR),
```

In the operational semantics, we interpret the instruction reader using `readMem`, picking up the memory component from the processor state.

Using `Reader`s that can be interpreted directly on the machine state improves somewhat on the approach of Myreen [21, Appendix A], in which the operational semantics always fetches 20 bytes from memory and checks for overflow only after decoding.

Instruction execution. Instruction execution makes significant use of the state monad for reading and writing registers, flags and memory. Through careful use of auxiliary definitions to capture commonality, our formalization is small and easily understood. Here is the interpretation of the `RET` instruction.

```
| RETOP offset ⇒
  let! oldSP = getRegFromProcState ESP;
  let! IP' = getDWORDFromProcState oldSP;
  do! setRegInProcState ESP
    (addB (oldSP+#4) (zeroExtend 16 offset));
  setRegInProcState EIP IP'
```

Putting it together. Given a machine state, the processor (a) decodes the bytes addressed by EIP to determine which instruction to execute; (b) advances EIP to the next instruction; and (c) executes the instruction. All this is captured by the following single-step transition function, written in monadic style.

```
Definition step : ST unit :=
  let! oldIP = getRegFromProcState EIP;
  let! (instr,newIP) = readFromProcState oldIP;
  do! setRegInProcState EIP newIP;
  evalInstr instr.
```

In addition to state, the processor monad incorporates exceptional behaviour which is used for actual processor exceptions such as division-by-zero and memory violations, but is also used to model unspecified behaviour. For example, the x86 specification states that flags `SF` and `PF` are *undefined* after executing the `MUL` instruction. We model this using an additional flag state `FlagUnspecified`, which if later scrutinized by a conditional branch instruction is interpreted as undefined behaviour, modelled by `evalInstr` returning the `None` value.

3. Assembling x86

A particular emphasis of our work on machine code verification is on using Coq as a place to do everything: modelling the machine, writing programs, assembling or compiling programs, and proving properties of programs. Coq's powerful notation feature makes it possible to write assembly programs, and higher-level language programs, inside Coq itself with no need for external tools.

3.1 Basics of assembly code

Syntax. At its most superficial, assembly code support means nice syntax for the `Instr` type, which is achieved using Coq's `Notation` and `Coercion` features. For example, the instruction `ADD EAX, [EBX + ECX*4 + 14]` considered earlier is valid syntax both in our Coq development and in real-world assemblers that make use of Intel-style syntax for x86.

Labels. An important aspect of assembly programs is the ability to define and reference named *labels*. To this end we define a type `program` to represent sequences of instructions, label scoping, label definition, and inline data:

```
Inductive program :=
  prog_instr (c: Instr)
| prog_skip | prog_seq (p1 p2: program)
| prog_declabel (body: DWORD → program)
| prog_label (l: DWORD)
| prog_data {T} {R: Reader T} {W: Writer T}
              (RT: Roundtrip R W) (v: T).
Coercion prog_instr: Instr >-> program.
Infix ";;" :=
  prog_seq
  (at level 62, right associativity).
Notation "'LOCAL' l ';' p" :=
  (prog_declabel (fun l ⇒ p))
  (at level 65, l ident, right associativity).
Notation "l ':;'" :=
  (prog_label l)
  (at level 8, no associativity, format "l ':;'").
```

The first three constructors just give us possibly-empty instruction sequences. The `prog_declabel` constructor and `LOCAL` notation introduces a new label name `l`, scoped within `p`. This use of Coq variables for object-level 'variables' (here, labels) is reminiscent of higher-order abstract syntax. The `prog_label` constructor (with familiar colon notation) is a pseudo-instruction whose address the assembler will assign to the label.

Here is an example of using labels to define a simple 'skip over' conditional:

```
(* Determine max(r1,r2), leaving result in r1 *)
Definition max (r1 r2: Reg) : program :=
  LOCAL Bigger;
    CMP r1, r2;; JG Bigger;; MOV r1, r2;;
  Bigger:; .
```

The final constructor `prog_data` packs a value together with round-tripping reader and writer; we can use this to define handy inline data directives, as used in the example of Figure 1. The ability of readers and writers to observe their current 'cursor' lets us define alignment directives similar to those supported by traditional assemblers:

```
Example exalign :=
LOCAL str; LOCAL num;
str:;; ds "Characters";;
num:;; align 2;; (* Align on 2^2 boundary i.e. DWORD *)
       dd #x"87654321". (* DWORD value *)
```

The assembler. To turn a `program` into a sequence of bytes suitable for execution, we must do two things: assign concrete addresses to `prog_declabel`-bound variables, and encode the instructions themselves.

For encoding, we create an instance of `Writer` for instructions, with various helper instances for auxiliary types used in instructions. Here is the fragment that deals with three variants of `push`:

```
Instance encodeInstr : Writer Instr := fun instr ⇒
  match instr with
  | PUSH (SrcI c) ⇒
    if signTruncate 24 (n:=7) c is Some b
    then do! writeNext #x"6A"; writeNext b
    else do! writeNext #x"68"; writeNext c

  | PUSH (SrcR r) ⇒
    writeNext (PUSHPREF ## injReg r)

  | PUSH (SrcM src) ⇒
    do! writeNext #x"FF";
    writeNext (#6, RegMemM true src)
```

As the reader will observe, `writeNext` is overloaded: in the first case, it is used to write a `BYTE` and then a `DWORD`, in the second case, we concatenate a constant prefix of five bits onto a three-bit encoding of registers, and in the third case, we use a `Writer` instance for the x86 *r/m32* addressing mode.

For label resolution, we use the classic two-pass approach. On the first pass over `program`, we instantiate each `prog_declabel` with a fresh `DWORD` that uniquely identifies the label, encode instructions only to obtain their size, and update a map from label declaration identifiers to addresses when encountering a `prog_label`. On the second pass, we instantiate `prog_declabel` with the calculated addresses and accumulate instruction encodings. Our current scheme assumes that label occurrences (for example, in branches) have fixed length encodings – a scheme that used more efficient variable-length encodings would iterate until label assignments reach a fixed point [5].

The assembler is packaged up as a `Writer program` that combines the passes in a clean monadic style, with `runWriter` used to compute the final sequence of generated bytes.

The assembler will fail if the instructions run off the end of memory or if the program uses labels in an ill-formed manner such as placing the same label twice.

Assembler correctness. No discussion of an assembler constructed within a proof assistant would be complete without some discussion of *proof*; and indeed, we have proved that the assembler does its job. First, we can prove that instruction encoding commutes with instruction decoding:

```
Instance RoundtripInstr : Roundtrip readInstr encodeInstr.
```

The correctness of the assembler itself is not a `Roundtrip` instance since there is no `Reader program`. There is instead an interpretation of `program` into separation logic [16], `interpProgram`, and we can prove correctness relative to this:

```
Theorem write_program_correct (i j: DWORD) (p: program):
  interpWriter i j p ⊢ interpProgram i j p.
```

This theorem is analogous to the `interpWriter_roundtrip` lemma from the end of Section 2.4: a program after assembly corresponds to its own logical interpretation.

3.2 Macros

A feature of most assemblers – though perhaps little used, now that most assembly code is machine-produced – is the provision of both built-in and programmer-defined *macros*.

Structured control. Built-in macros are used, for example, to build conditionals and loops without going through the pain of declaring labels and branching. In Coq, simple definitions, together with scoped labels and judicious use of notation, make this very easy. Here, for example, is a conditional construct that tests whether flags according to condition code `cond` have boolean value `value`, and executes `pthen` or `pelse` accordingly.

```
Definition ifthenelse (cond: Condition) (value: bool)
  (pthen pelse: program) : program :=
  LOCAL THEN; LOCAL END;
    JCC cond value THEN;;
      pelse;; JMP END;;
    THEN:;; pthen;;
    END:;.
```

A while-loop can be defined similarly, this time incorporating test code `ptest` that sets the flags appropriately. (See Figure 1 for an example of its use.)

```
Definition while (ptest: program)
  (cond: Condition) (value: bool)
  (pbody: program) : program :=
  LOCAL BODY; LOCAL test;
    JMP test;;
  BODY:;; pbody;;
  test:;;
    ptest;;
    JCC cond value BODY.
```

Such definitions really shine when proving correctness of machine code programs using our separation logic framework [16], as we can give derived Hoare-style proof rules for the macros.

Procedures. We can also devise macros to package up procedures and their calling conventions. For example, Figure 5 codes a non-standard 'leaf' calling convention in which the return address is simply stored in the register `EDI`, together with some notation for locally-scoped procedure declarations. Earlier we saw an application of this in the factorial example of Figure 1. We consider this further in Section 4.2.

4. More substantial examples

4.1 Multiplication by a constant

Using Coq definitions, possibly involving recursion, we can write macros that are akin to the instruction selection phase of a compiler. For example, the function below generates shift and addition instructions to compute $r_1 := r_1 + r_2 \cdot m$ for some constant m. (In the good old days, this would have been more efficient than using the processor's own multiplication instruction. Alas, this is probably no longer the case, but hopefully it illustrates the power of macros.)

```
Fixpoint add_mulc nbits (r1 r2: Reg) (m: nat) :=
  if nbits is nbits'.+1
  then if odd m
       then ADD r1, r2;;
            SHL r2, 1;;
```

```
Definition callproc f :=
  LOCAL iret;
   MOV EDI, iret;; JMP f;;
  iret:;.

Definition defproc (p: program) :=
  p;; JMP EDI.

Notation "'letproc' f ':=' p 'in' q" :=
  (LOCAL skip; LOCAL f;
    JMP skip;;
   f:;;   defproc p;;
   skip:;; q)
  (at level 65, f ident, right associativity).

(* Multiply EAX by nine, trashing EBX *)
Example ex :=
  letproc tripleEAX :=
    MOV EBX, EAX;; SHL EAX, 2;; ADD EAX, EBX
  in
    callproc tripleEAX;; callproc tripleEAX.
```

Figure 5. Example: procedure macros

```
          add_mulc nbits' r1 r2 m./2
      else SHL r2, 1;;
          add_mulc nbits' r1 r2 m./2
  else prog_skip.
```

Of course, one could imagine doing something similar in a high-level language such as Haskell. The crucial difference is that we can also state and prove correctness of a specification for the macro:

```
Lemma add_mulcCorrect nbits : ∀ (r1 r2: Reg) m,
  m < 2^nbits →
  ⊢  ∀ v, ∀ w,
  basic
  (r1 ↦ v ** r2 ↦ w ** OSZCP_Any)
  (add_mulc nbits r1 r2 m)
  (r1 ↦ addB v (mulB w (fromNat m)) ** r2? ** OSZCP_Any).
```

The syntax following $\vdash$ is an expression in our specification logic [16]. It states that `add_mulc bits nbits r1 r2 m` behaves as a block of code with single entry and exit points, and can be specified by a `basic` Hoare-style triple, whose precondition gives initial values `v` and `w` for registers `r1` and `r2`, assumes that the five standard flags have arbitrary values, and whose postcondition assigns the appropriate result to register `r1` and says nothing about `r2` and the flags. Notice the use of separating conjunction, written as `**`; in this specification it serves merely to ensure that `r1` and `r2` are distinct.

Having proved a crisp specification for `add_mulc` we can use it from within subsequent code and their proofs of correctness just as if there existed a special add-and-multiply instruction.

More complex and cunning use of x86 instructions can be encapsulated in such definitions. So, for example, the macro instantiation

```
add_mulcFast EDI EDX #160
```

reduces to

```
SHL EDX, 5;;
ADD EDI, EDX;;
LEA EDI, [EDI + EDX * 4 + 0]
```

that uses x86 address arithmetic in addition to a shift operation. The specification of `add_mulcFast` is identical to that of the less efficient `add_mulc`.

4.2 Calling conventions

To some extent it's possible to hide details of calling conventions using macros: we already saw an example of this in Figure 1, where the calling sequence for a three-argument `cdecl` function call was packaged up as a Coq definition. More interestingly, we can *generate* caller and callee boilerplate code, and its specification, from descriptions of calling conventions.

We consider three standard x86 calling conventions, described using an enumeration type:

```
Inductive CallConv := cdecl | stdcall | fastcall.
```

We then write a function `callconv` that takes takes a calling convention and arity, and returns a pair `(call, def)` in which `call` is a pseudo-instruction that expands to the calling sequence for that convention and number of arguments, and `def` wraps a function body with appropriate prologue and epilogue, and supplies argument accessors. The type of `callconv` is mildly dependent, in that the types of `call` and `def` vary according to the specified arity.

Here it is used to define and call a function that computes the sum of its three arguments:

```
Example addfun (cc: CallConv) :=
  let (call, def) := callconv cc (mkFunSig 3 true) in
  LOCAL MyFunc;
  MyFunc:;; def (fun arg1 arg2 arg3 ⇒
    MOV EAX, arg1;;
    ADD EAX, arg2;;
    ADD EAX, arg3);;
  call MyFunc 2 3 4.
```

Let's see what happens when we instantiate `addfun` with the three standard calling conventions. First, with `cdecl`, in which arguments are passed on the stack, right-to-left, and the caller has to clean up the stack:

```
LOCAL MyFunc;
MyFunc:;;
  PUSH EBP;; MOV EBP, ESP;; (* prologue *)
  MOV EAX, [EBP + 8];;
  ADD EAX, [EBP + 12];;
  ADD EAX, [EBP + 16];;
  POP EBP;; RET 0;;       (* epilogue *)
PUSH 4;; PUSH 3;; PUSH 2;;
CALL MyFunc;; ADD ESP, 12
```

Second, with `stdcall`, in which arguments are again passed on the stack, right-to-left, but the callee cleans up the stack:

```
LOCAL MyFunc;
MyFunc:;;
  PUSH EBP;; MOV EBP, ESP;; (* prologue *)
  MOV EAX, [EBP + 8];;
  ADD EAX, [EBP + 12];;
  ADD EAX, [EBP + 16];;
  POP EBP;; RET 12;;      (* epilogue *)
PUSH 4;; PUSH 3;; PUSH 2;;
CALL MyFunc
```

Lastly, with `fastcall`, in which the first two arguments are passed in registers ECX and EDX:

```
LOCAL MyFunc;
MyFunc:;;
  PUSH EBP;; MOV EBP, ESP;;
  MOV EAX, ECX;;
  ADD EAX, EDX;;
  ADD EAX, [EBP + 8];;
  POP EBP;; RET 4;;
PUSH 4;; MOV ECX, 2;; MOV EDX, 3;;
CALL MyFunc.
```

Compared to earlier work on formalization of calling conventions [22], the potential of a proof assistant such as Coq is in generating *specifications* from calling convention descriptions. We hope to develop this idea further, even using the *types* in C-style signatures to generate more precise specifications for calls to external libraries whose implementation is not under our control.

4.3 Regular expressions

Writing large amounts of raw assembly code is tedious and difficult, and verifying it even more so. To produce non-trivial programs, we clearly want to move up the abstraction stack for both programming and proving as quickly as possible. The kind of macros we have shown so far take the first step in that direction; the next is to implement more self-contained domain-specific languages.

Coq, and dependently-typed languages in general, offer an ideal environment for embedding domain-specific languages (DSLs) [6, 9, 23]. Dependent types support the type-safe embedding of object languages [4] and Coq's mixfix notation system enables reasonably idiomatic domain-specific concrete syntax. The assembler, and Coq's powerful abstraction facilities, potentially provide a flexible framework in which to not only implement and verify a range of *domain-specific compilers* [12, 24], but also to combine them, and reason about their combination. Working with many DSLs optimizes the "horizontal" compositionality of systems, and favours reuse of building blocks, by contrast with the "vertical" composition of the traditional compiler pipeline, involving a stack of comparatively large intermediate languages that are harder to reuse the higher one goes.

The idea of building compilers from reusable building blocks is a common one, of course. But the interface contracts of such blocks tend to be complex, so combinations are hard to get right. We believe that being able to write and verify formal specifications for the pieces will make it possible to know when components can be combined, and should help in designing good interfaces. Furthermore, the fact that Coq is also a system for formalizing mathematics enables one to establish a close, formal connection between embedded DSLs and non-trivial domain-specific *models*. The possibility of developing software in a *truly* 'model-driven' way is an exciting one.

As a small example, we present in this section a certified compiler from regular expressions to x86 machine code. We make crucial use an existing Coq formalization, due to Braibant and Pous [7], of the theory of Kleene algebras. We do not merely use the theory developed in that library to prove correctness of our little compiler, but actually reuse the Coq-executable translation from regular expressions to deterministic finite-state automata (DFAs) that is part of their formalization. We write and certify a compiler from DFAs to machine code, which composes with the library's certified translation to yield a verified compiler for regular languages. That this works despite the fact that the Kleene algebra formalization was developed with no thought of implementation is a nice illustration of the power of constructive mathematics.

We later apply the resulting pipeline to recognize strings representing floating-point numbers. One could certainly write such a particular validator in assembly. However, proving the correctness of a special instance is likely to be as laborious as proving the correctness of the more generally useful regular expression compiler.

DFAs: We generate machine code from DFAs, which have the following components:

```
Variables
    (alphabet: list DWORD)
    (dfa_size : nat)
    (dfa_init: dfa_state)
    (trans: dfa_state → DWORD → dfa_state)
    (accept: dfa_state → bool).
```

where `dfa_state` is `'I_dfa_size`, the SSReflect finite type of naturals less than `dfa_size`, and `alphabet` specifies the allowable subset of `DWORD`. The initial state of the automaton is `dfa_init` while the accepting states are specified by `accept`. The transition function is coded by `trans`.

The language accepted by a DFA is defined inductively over an input word, i.e. a list of `DWORD`s:

```
Fixpoint lang (s: dfa_state) (w: seq DWORD): bool :=
  if w is a::w'
  then (a \in alphabet) && lang (trans s a) w'
  else accept s.
```

We say that the word w is accepted by the automaton if

$$\mathsf{lang}\ \mathsf{dfa\_init}\ w \equiv \mathsf{true}$$

Compiling DFAs: From a DFA we generate code that processes a 0-terminated string of `DWORD`s, stored in memory at some address `buffer`. In our logic, we write `buffer :->0 w`, for `w` a sequence of non-null double words, for this precondition. The compiler creates a tuple of assembler labels

```
labels : dfa_size.-tuple DWORD
```

that are in one-to-one correspondence with the states of the automaton. Each label corresponds to the entry-point of the transition function at that state; the label associated with a DFA state `s` is obtained by the tuple-lookup `[tnth labels s]`.

The key component of code generation is the compilation of the transition function at a single state `s`. Besides potentially jumping to the labels associated with other states, each transition function can take two possible "continuations": it can either jump to address `acc` to accept a word, or to address `rej` to reject a word.

Compiling a transition: The compilation of a transition is presented in Figure 6. `EAX` points to a symbol, initially the first element of the buffer. If the current character is the terminating symbol `0`, we take the accepting or rejecting continuation, depending on whether the current state is accepting or not. Note that metaprogramming is at play here: the `if` statement is evaluated in Coq, at compile time.

If the current character is non-zero, it is processed through a sequence of conditional branches: if any character of the alphabet matches, the code jumps to the appropriate state label according to the DFA's transition function. Again, the jump table is computed in Coq. If we reach the end of the jump table, the character in memory is not in the alphabet so the word is rejected. The resulting code is therefore a rather direct assembler translation of the function `trans s`.

Compiling the DFA: The code for the whole DFA is obtained by iterating over all states, concatenating the compiled version of the individual transition functions. Because each snippet generated by `transition s` starts with the label `s`, the assembler will resolve these names automatically to the desired addresses. To start the automaton, we simply jump to the initial state:

```
Definition DFA_to_x86: program :=
  JMP (tnth labels dfa_init);;
  foldr prog_seq prog_skip
        [tuple transition labels s
        | s < dfa_size].
```

Generating the DFA: Braibant and Pous's ATBR library [7] exports a datatype `regex` of regular expressions, with constructors for the empty set, the empty word, literals, concatenation, and star operators. They also define their own type `DFA.t` of DFAs, operating over sequences of Coq's `positives` and a *computable* function from regular expressions to their automata:

```
Definition transition : program :=
  (tnth labels s):;;

    (* Move pointer to next character *)
    MOV EBX, [EAX] ;;
    ADD EAX, (#4 : DWORD) ;;

    (* Accept/reject if end of string: *)
    CMP EBX, (#0: DWORD) ;;
    JE (if accept s then acc else rej);;

    (* Jump table: *)
    foldr prog_seq prog_skip
          [seq CMP EBX, (c: DWORD) ;;
               JE (tnth labels (trans s c))
          | c <- alphabet] ;;

    (* Not in alphabet: *)
    JMP rej.
```

Figure 6. Compilation of a transition

```
Lemma DFA_to_x86_correct
       (w: seq DWORD) :

    ⊢  ∀ l,

    (   (safe ⊗    (lang dfa_init w
                ∧  EIP ↦▷ acc) ⊗ EAX?)
    ∧   (safe ⊗ (¬  lang dfa_init w
                ∧  EIP ↦▷ rej) ⊗ EAX?)
   (* ---------------------------------*)
    ⟶  safe ⊗ EIP ↦▷ l ⊗ EAX ↦▷ buffer )

   (* Memory: *)
     ⊗ EBX?
     ⊗ buffer ↦^0 w

   (* Program: *)
     ⊘ l ↦  DFA_to_x86.
```

Figure 7. Correctness of the DFA compiler

```
Definition X_to_DFA (a: regex): DFA.t := (...)
```

Given a regex `r`, we construct the components of one of our DFAs from `X_to_DFA r`; this is morally the identity, but for the translation from non-zero `DWORD`s to `positive`. Applying `DFA_to_x86` to these components, `buffer`, `acc` and `rej` then yields an x86 program that recognizes the original regular expression.

Correctness of DFA compilation: The correctness of DFA compilation follows naturally from the close correspondence between execution of the source DFA and the target code. Where the DFA takes a transition from one state to another upon recognizing a character, the machine jumps to the associated label. Where the DFA accepts or rejects a word based on the status of the final state, the machine jumps to `acc` or `rej`, the choice of which depends on whether the state is accepting or not.

The formal statement of the correctness of DFA compilation is shown in Figure 7. The generated code is unstructured, being a mutually-recursive collection of blocks, each of which can call the others or jump out to either of the exit labels `acc` and `rej`. The traditional Hoare-triple expressed by `basic`, which only allows for a single precondition and a single postcondition, is not therefore appropriate. We instead use a more primitive continuation-passing style of specification [2, 16, 26]: *if* it is safe to jump to the two exits ($\mathsf{EIP} \mapsto_{\triangleright} acc$ and $\mathsf{EIP} \mapsto_{\triangleright} rej$, with appropriate conditions holding), *then* it is safe to jump to the entry point of the compiled code ($\mathsf{EIP} \mapsto_{\triangleright} l$).

We work under a 'read-only' frame [16], which specifies the preservation of the program code located at address `l`, and two frames, the first stating the existence of the `EBX` flag and another stating the existence of a null-terminated string `w` at address `buffer`. In this context, it is safe to execute the code at address `l` with `EAX` pointing to the buffer – expressed by $\mathsf{safe} \otimes \mathsf{EIP} \mapsto_{\triangleright} l \otimes \mathsf{EAX} \mapsto_{\triangleright} buffer$ – provided that:

- it is safe to take the continuation `acc`, formally

  $$\mathsf{safe} \otimes (\ldots) \wedge \mathsf{EIP} \mapsto_{\triangleright} acc$$

  when the word in memory is accepted by the DFA, formally

  $$\mathsf{lang\ dfa\_init}\ w$$

- it is safe to take the continuation `rej`, formally

  $$\mathsf{safe} \otimes (\ldots) \wedge \mathsf{EIP} \mapsto_{\triangleright} rej$$

  when the word in memory is *not* accepted by the DFA, formally

  $$\neg(\mathsf{lang\ dfa\_init}\ w)$$

That is, the x86 code takes the success (respectively, failure) continuation if and only if the DFA accepts (respectively, rejects) the word `w`. Note that we only enforce *partial* correctness: the infinite loop is a valid implementation of this specification.

The correctness proof for the compiler relies on a Bekič-like lemma that uses the Löb rule to establish safety for sets of mutually recursive code fragments. We proved the specification by first showing simple memory safety and then straightforwardly extending the specifications with the logical invariants concerning language acceptance. Note that this specification requires the original string to be preserved in memory, as the computation is framed under the buffer containing `w`. A small modification yields a version that would also be satisfied by an implementation that read the data destructively.

Correctness of regular expression compilation: Braibant and Pous prove [7] three results that together imply that the language accepted by a regular expression `r` and the language accepted by the `DFA.t` (`X_to_DFA r`) coincide:

```
Lemma X_to_DFA_correct:
    ∀ a, DFA.eval (X_to_DFA a) == a.

Lemma X_to_DFA_bounded: ∀ a, DFA.bounded (X_to_DFA a).

Theorem language_DFA_eval: ∀ A, DFA.bounded A →
    regex_language (DFA.eval A) == DFA_language A.
```

Composing these results with a lemma stating that `lang dfa_init` coincides with their `DFA_language (X_to_DFA r)` (on strings from `alphabet`), we can rewrite the correctness theorem of Figure 7 using the following equivalence, for `w` a sequence of `DWORD`s containing no zeros:

```
lang dfa_init w ↔
        regex_language r (map char_of_DWORD w)
      ∧ (all (fun a ⇒ a \in alphabet) w).
```

to get a fully verified pipeline for regular expression compilation. Further, the specification is then compatible with the rest of ATBR, potentially allowing more complex language-theoretic results about the behaviour of the code to be derived.

Matching floating-point numbers: As a small test, we exercised our pipeline on the following regular expression

```
Definition FP: regex :=
  [[ "-" , "+"]]? ’
  [{ "0" , "9" }]* ’ $"." ? ’
  [{ "0" , "9" }]+ ’
  ($"e" ’ [[ "-" , "+"]]? ’ [{ "0" , "9" }]+)?.
```

that recognizes strings representing signed floating-point numbers. Note the use of Coq notation to provide an intuitive syntax for defining regular expressions.

The intermediate DFA consists of 12 states over an alphabet of 14 letters. This results in 2321 bytes of machine code. We have packaged this code into an executable that retrieves a string from the standard input (through a system call to `gets`), run the DFA on that string, and reports its status through the `puts` system call. Adding the (currently unverified) wrapper yields a 4K-byte binary.

5. Discussion and related work

There is a long tradition of using proof assistants to formalize processors and verify low-level programs. Notable early work includes the verification in the Boyer-Moore prover of the the Piton assembler for the verified FM8502/9001 microprocessors [18]. For reasons of space, we here discuss just a few more recent pieces of related work.

Tuch et al. [28] describe an Isabelle/HOL formalization in which a C-language view of memory as a collection of typed objects is layered over an underlying total map from addresses to bytes. Isabelle's type classes are used to infer the byte-encodings of different types and a ghost heap type description allows separation between the representations of distinct types to be inferred and maintained fairly automatically. A further separation logic layer is used to deal with non-aliasing between values of the same C language type.

Mulligan and Sacerdoti Coen [20] have used Matita to formalize the MCS-51 microcontroller and the correctness of an assembler for it. The assembler uses pseudoinstructions for jumps to labels, which may be expanded into different kinds of branch in the object code. Correctness is then presented as a simulation between the operational semantics of object and assembly programs, but the result is conditional on the program (dynamically) not manipulating addresses in ways judged to be ill-behaved (e.g. performing arithmetic on them), and on the correctness of the branch displacement decisions. Our assembler does not currently make branch decisions; we plan to address this at a slightly higher level of abstraction, giving a semantic (rather than syntactic) interpretation to a low-level type system distinguishing pointers from more concrete data.

The targets of the CompCert compiler [17] are comparatively high-level assembly languages for PowerPC, ARM and x86. The operational semantics of these assembly languages are defined over a C-like memory model (rather than the lower-level array of bytes view we take here), and there are pseudoinstructions for label, allocation and stack operations. CompCert's treatment of machine integers is less pervasively dependent than ours, exploiting Coq's module system to provide operations on integers of particular sizes, and using Coq's arbitrary precision integers, together with proofs that they are within a certain range, rather than raw sequences of bits.

RockSalt [19] is a verified checker for the sandboxing policy used in Google's Native Client. The verification of RockSalt relies on a Coq model of x86, built using two domain specific languages. One is a regular expression parser, used to decode bitstrings in memory into an inductive datatype of x86 instructions. The abstraction provided by this DSL is similar to that of our reader monads, though non-determinism in the grammars makes establishing the determinism of decoding somewhat involved. The second DSL is a register transfer language, used to define the transition function for decoded instructions, which plays a similar role to the imperative monadic language we use for the same purpose. Our embeddings are shallow, however, whilst those used in RockSalt are deep (i.e. there are inductive datatypes for both grammars and RTL instructions). RockSalt uses CompCert's libraries for machine integers.

Chlipala has built a model of x86 in Coq for the purpose of verifying OCaml-extractable verifiers for machine code [8]. In this work, the Coq functions work on an assembly-level type of instructions, with binary decoding being delegated to an OCaml function. His Bedrock framework [10] for programming and verifying (with an impressive degree of automation) low-level programs works with a rather idealized assembly language. Like our system, Bedrock makes good use of Coq's customizable syntax to allow low-level programs to be written in a convenient way within the proof assistant, including user-defined macros. Bedrock's program syntax also embeds pre- and post-conditions and hints to the proof automation.

Fox and Myreen have formalized the ARMv7 ISA in HOL4 [13]. This is an unusually comprehensive formalization, presented in a monadic style somewhat like ours, and has been subjected to testing against real hardware. Myreen has also built a model of x86 machine code and used it to verify a JIT compiler for a simple language [21]. This model is notable for carefully modelling the x86 instruction cache and treating both encoding and decoding of instructions.

There are many similarities between these projects and the work described here, but none have our focus on making it convenient to write, compile and verify the code that actually runs, relative to a model of the underlying hardware, entirely within one system.

This paper has focused mainly on the modelling and programming aspects, as our earlier paper [16] describes the program logic in some detail. Many features of Coq on which we rely for these first two tasks have analogues in other systems. Haskell and Isabelle both feature type classes and, whilst not dependently typed, can simulate some of our uses of dependency (e.g. using type-level naturals to express n-bit words), and the use of HOAS for representing object level binding is common in DSLs embedded in functional languages. Full dependent types are more powerful and straightforward than clever encodings in Haskell (and we use them extensively), but the main advantage of Coq is certainly the close integration with verification. We believe that such tight integration may partially compensate for the fact that languages embedded in a system like Coq will inevitably be more 'clunky' than ones with their own custom parser, type checker, error messages and so on.

For producing trustworthy software, proof assistants are the only tools in which *all* the artefacts in which we are interested (programs, languages, models, specifications, proofs, ...) can coexist and be formally related. Modern software components are often comparatively small and specialized, but must meet specifications that are inherently rather rich, in that they involve non-trivial mathematical structures that cannot be directly expressed and reasoned about in the logics of traditional verification tools. A fully integrated approach suggests that one could construct systems with strong correctness guarantees by combining verified components written in several domain-specific languages, each with its own domain-specific metatheory. Our regular expression compiler provides a simple example of how a piece of formalized mathematics (not originally designed for use in generating or proving machine code) can be used in verification; some other compiler correctness results have a similar flavour [3, 21]. Of course, just having all the components programmed in one metalanguage does not magically make them compatible, but having them accompanied by specifications makes it possible to both say what compatibility means, and to prove properties of combined systems in the case that the components *are* compatible.

Promising though such ideas are, the combination of Coq, Proof General and Emacs is, we must admit, still some way from being the multilanguage integrated development and verification environment of our dreams. Predictable things we would like include better techniques for dealing with object language binding and integration with external solvers (though we could certainly do much more in terms of automation entirely within Coq). Nor has our experience with doing all computation within Coq itself been entirely trouble-free. Though this is intellectually appealing and (arguably) reduces the TCB, computation within Coq is inherently less efficient than extracting and compiling to native code, internal terms can be large, and additional time and space is used reducing non-computational parts of them. More critically, it is all too easy to make definitions that do not compute at all; this is a common problem when using the SSReflect libraries, which use opaque definitions to improve efficiency of proving. Avoiding such issues, particularly when using third-party libraries, still feels like something of a black art. It remains to be seen to what extent our current approach to computation can really scale, but Coq continues to improve in this regard, and we always have the option of extraction to fall back on.

Finally, we note an interesting phenomenon. The requirements of verification tend to push us towards writing even small fragments of assembly code in a surprisingly abstracted, modular style. For example, a single increment instruction used to step through a string is naturally abstracted as the 'next' function of a much more generic abstract iterator interface, just because that's the best way to structure the associated proof. On the one hand, such refactorings feel like good software engineering and really do make the surrounding verified code usefully more generic; on the other, they do make programs very 'bitty', as one rarely writes more than a handful of instructions without wanting to abstract something one would not (or possibly could not) have abstracted without the requirements (or possibility) of verification. Whether this should be counted as a benefit or drawback of our approach, we are as yet undecided.

References

[1] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *LNCS*. Springer, 2005.

[2] N. Benton. A typed, compositional logic for a stack-based abstract machine. In *3rd Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *LNCS*, 2005.

[3] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *14th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2009.

[4] N. Benton, C-K. Hur, A. J. Kennedy, and C. McBride. Strongly typed term representations in Coq. *J. Automated Reasoning*, 49, 2011.

[5] J. Boender and C. S. Coen. On the correctness of a branch displacement algorithm. *Preprint arXiv:1209.5920*, 2012.

[6] E. Brady and K. Hammond. Resource-safe systems programming with embedded domain specific languages. In *14th International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 7149 of *LNCS*. Springer, 2012.

[7] T. Braibant and D. Pous. An efficient Coq tactic for deciding Kleene algebras. In *1st International Conference on Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*. Springer, 2010.

[8] A. Chlipala. Modular development of certified program verifiers with a proof assistant. *J. Functional Programming*, 18(5/6), 2008.

[9] A. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2010.

[10] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011.

[11] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2 (2A, 2B & 2C): Instruction Set Reference*, January 2013.

[12] P.-E. Dagand, A. Baumann, and T. Roscoe. Filet-o-Fish: practical and dependable domain-specific languages for OS development. *SIGOPS Oper. Syst. Rev.*, 43(4):35–39, 2010.

[13] A. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instuction set architecture. In *1st International Conference on Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*. Springer, 2010.

[14] G. Gonthier. Advances in the formalization of the odd order theorem. In *2nd International Conference on Interactive Theorem Proving (ITP)*, volume 6898 of *LNCS*. Springer, 2011.

[15] G. Gonthier, A. Mahboubi, and E. Tassi. A small scale reflection extension for the Coq system. Technical Report 6455, INRIA, 2011.

[16] J. B. Jensen, N. Benton, and A. J. Kennedy. High-level separation logic for low-level code. In *40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2013.

[17] X. Leroy. Formal verification of a realistic compiler. *Comm. ACM*, 52 (7):107–115, 2009.

[18] J. Strother Moore. *Piton: A Verified Assembly-Level Language*. Kluwer, 1996.

[19] G. Morrisett, G. Tan, J. Tassarotti, J.B. Tristan, and E. Gan. Rocksalt: Better, faster, stronger SFI for the x86. In *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2012.

[20] D. P. Mulligan and C. Sacerdoti Coen. On the correctness of an assembler for the MCS-51 microprocessor. In *2nd International Conference on Certified Programs and Proofs (CPP)*, volume 7679 of *LNCS*. Springer, 2012.

[21] M.O. Myreen. Verified just-in-time compiler on x86. In *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2010.

[22] R. Olinsky, C. Lindig, and N. Ramsey. Staged allocation: A compositional technique for specifying and implementing procedure calling conventions. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2006.

[23] N. Oury and W. Swierstra. The power of pi. In *13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2008.

[24] L. Pike, N. Wegmann, S. Niller, and A. Goodloe. Experience report: a do-it-yourself high-assurance compiler. In *17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2012.

[25] B. Spitters and van der Weegen. E. Type classes for mathematics in type theory. *Math. Structures in Comp. Sci.*, 21:1–31, 2011.

[26] G. Tan and A. W. Appel. A compositional logic for control flow. In *7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *LNCS*, 2006.

[27] The Coq Development Team. *The Coq Reference Manual, version 8.4*, 2012. URL `http://coq.inria.fr`.

[28] H. Tuch, G. Klein, and M. Norrish. Types, bytes and separation logic. In *34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2007.