Adventures in Verifying Arithmetic

John Harrison

Amazon Web Services

28th May 2020 (14:00-15:00 Austin, 12:00-13:00 Pacific)

- From arithmetic on $\mathbb{R}$ to arithmetic on $\mathbb{Z}$
  - Floating-point verification at Intel
  - Crypto bignum verification at AWS
- Points of similarity, points of contrast
  - Requirements (correctness, efficiency, security)
  - Formalizing mathematics, continuous and discrete
  - Programming custom inference rules
  - The general usefulness of interval bounds
  - Newton's method vs. Hensel lifting
  - ISA modeling and continuous integration
- Conclusions

2006: Verifying floating-point arithmetic at Intel

2019: Verifying crypto bignums at AWS

Floating-point kernels v cryptographic primitives

- They are *both* intended to be mathematically correct (give the right answer or 'within 0.52 ulps')

Floating-point kernels v cryptographic primitives

- They are *both* intended to be mathematically correct (give the right answer or ‘within 0.52 ulps’)
- They are *both* intended to be fast

Floating-point kernels v cryptographic primitives

- They are *both* intended to be mathematically correct (give the right answer or 'within 0.52 ulps')
- They are *both* intended to be fast
- Crypto bignums often need to be *constant-time* (to avoid timing side-channels), and this may take precedence over average-case speed

Floating-point kernels v cryptographic primitives

- They are *both* intended to be mathematically correct (give the right answer or 'within 0.52 ulps')
- They are *both* intended to be fast
- Crypto bignums often need to be *constant-time* (to avoid timing side-channels), and this may take precedence over average-case speed

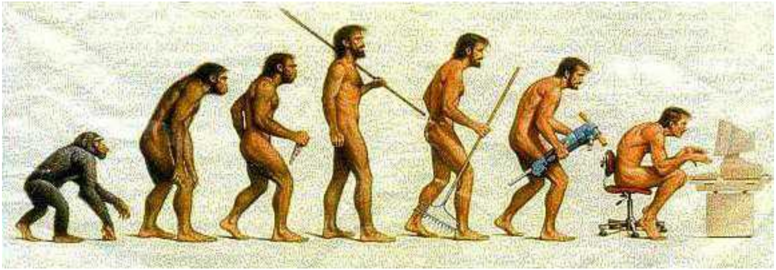
For this collection of reasons, we are writing and verifying code at the machine code level.

From arithmetic on $\mathbb{R}$ to arithmetic on $\mathbb{Z}$

Moving from the richer and more sophisticated real number system to integer arithmetic (and modular arithmetic at that).

From arithmetic on $\mathbb{R}$ to arithmetic on $\mathbb{Z}$

Moving from the richer and more sophisticated real number system to integer arithmetic (and modular arithmetic at that).
Looks like regressive evolution!



Mathematical contrasts

- The mathematical structure $\mathbb{R}$ of reals is a richer field containing the integers $\mathbb{Z}$ as a subring. But in practice we are interested in some finite subsets.

Mathematical contrasts

- The mathematical structure $\mathbb{R}$ of reals is a richer field containing the integers $\mathbb{Z}$ as a subring. But in practice we are interested in some finite subsets.
- Floating-point numbers can be considered as a subset of $\mathbb{R}$ but operations have more intricate mathematical properties
  - Most everyday algebraic laws like $x + (y + z) = (x + y) + z$ fail, though commutativity is more or less true (except for NaNs)
  - Rounding is a fundamentally important operation, with some regular properties but also many difficulties

Mathematical contrasts

- The mathematical structure $\mathbb{R}$ of reals is a richer field containing the integers $\mathbb{Z}$ as a subring. But in practice we are interested in some finite subsets.
- Floating-point numbers can be considered as a subset of $\mathbb{R}$ but operations have more intricate mathematical properties
  - Most everyday algebraic laws like $x + (y + z) = (x + y) + z$ fail, though commutativity is more or less true (except for NaNs)
  - Rounding is a fundamentally important operation, with some regular properties but also many difficulties
- In cryptography, we are mainly concerned with operations on $\mathbb{Z}_n$, the integers modulo n. This is at least a ring, and if n is prime it's a field (multiplicative inverses exist).

Mathematical similarities

There are meaningful analogies between ‘metrical’ and ‘p-adic’ algorithms:

- Over $\mathbb{R}$ where *things get smaller*
- Over $\mathbb{Z}$ where *things get more divisible by something*

Mathematical similarities

There are meaningful analogies between 'metrical' and 'p-adic' algorithms:

- Over $\mathbb{R}$ where *things get smaller*
- Over $\mathbb{Z}$ where *things get more divisible by something*

Nice table from Brent and Zimmermann "Modern Computer Arithmetic".

2.1.4 MSB vs LSB algorithms

Many classical (most significant bits first or MSB) algorithms have a p-adic (least significant bits first or LSB) equivalent form. Thus several algorithms in this chapter are just LSB-variants of algorithms discussed in Chapter 1 – see Table 2.1 below.

classical (MSB)	p-adic (LSB)
Euclidean division	Hensel division, Montgomery reduction
Svoboda's algorithm	Montgomery–Svoboda
Euclidean gcd	binary gcd
Newton's method	Hensel lifting

Table 2.1 *Equivalence between LSB and MSB algorithms*.

A common tool: HOL Light

- HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.

A common tool: HOL Light

- HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.
- An LCF-style proof checker for classical higher-order logic built on top of (polymorphic) simply-typed λ-calculus.

A common tool: HOL Light

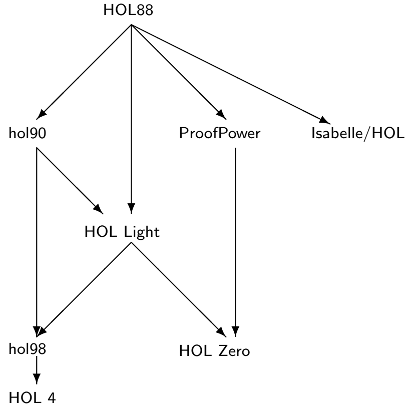
- HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.
- An LCF-style proof checker for classical higher-order logic built on top of (polymorphic) simply-typed λ-calculus.
- HOL Light is designed to have a particularly simple and clean logical foundation.

A common tool: HOL Light

- HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.
- An LCF-style proof checker for classical higher-order logic built on top of (polymorphic) simply-typed λ-calculus.
- HOL Light is designed to have a particularly simple and clean logical foundation.
- Written in Objective CAML (OCaml), a somewhat popular variant of the ML family of languages.

The HOL family DAG

There are many HOL provers, of which HOL Light is just one, all descended from Mike Gordon's original HOL system in the late 1980s.



Why HOL Light?

We need a general theorem proving system with:

- High standard of logical rigor and reliability
- Ability to mix interactive and automated proof
- Programmability for domain-specific proof tasks
- A substantial library of pre-proved mathematics

Needless to say ACL2 has also been used in these and similar domains, as have Coq, HOL4, Isabelle/HOL, PVS etc.

Formalizing mathematics

For floating-point verifications the mathematics required is mostly:

- Elementary number theory and real analysis
- Floating-point numbers, results about rounding etc.

Formalizing mathematics

For floating-point verifications the mathematics required is mostly:

- Elementary number theory and real analysis
- Floating-point numbers, results about rounding etc.

For the crypto bignums

- Additional number theory (e.g. Miller-Rabin pseudoprimes)
  `github.com/jrh13/hol-light/blob/master/Examples/miller_rabin.ml`

Formalizing mathematics

For floating-point verifications the mathematics required is mostly:

- Elementary number theory and real analysis
- Floating-point numbers, results about rounding etc.

For the crypto bignums

- Additional number theory (e.g. Miller-Rabin pseudoprimes)

  `github.com/jrh13/hol-light/blob/master/Examples/miller_rabin.ml`

- Elementary group theory, properties of elliptic curve groups

  `github.com/jrh13/hol-light/blob/master/Examples/nist_curves.ml`

Custom inference rules

For floating-point verifications:

- Verifying solution set of some quadratic congruences
- Proving primality of particular numbers
- Verifying error bounds in polynomial approximations

Custom inference rules

For floating-point verifications:

- Verifying solution set of some quadratic congruences
- Proving primality of particular numbers
- Verifying error bounds in polynomial approximations

For crypto bignums

- Proving equational theorems in abstract groups and rings
- Reasoning about general properties of congruences

Automating divisibility reasoning

Linear (Presburger) arithmetic is a common workhorse in formal verifications. For a lot of the 'congruential' reasoning a custom decision procedure is a similarly useful workhorse:

$$d|a \wedge d|b \Rightarrow d|(a-b)$$
$$\mathrm{coprime}(d,a) \wedge \mathrm{coprime}(d,b) \Rightarrow \mathrm{coprime}(d,ab)$$
$$\mathrm{coprime}(d,ab) \Rightarrow \mathrm{coprime}(d,a)$$
$$\mathrm{coprime}(a,b) \wedge x \equiv y \pmod{a} \wedge x \equiv y \pmod{b} \Rightarrow x \equiv y \pmod{(ab)}$$
$$m|r \wedge n|r \wedge \mathrm{coprime}(m,n) \Rightarrow (mn)|r$$
$$\mathrm{coprime}(xy, x^2+y^2) \Leftrightarrow \mathrm{coprime}(x,y)$$
$$\mathrm{coprime}(a,b) \Rightarrow \exists x.\ x \equiv u \pmod{a} \wedge x \equiv v \pmod{b}$$
$$ax \equiv ay \pmod{n} \wedge \mathrm{coprime}(a,n) \Rightarrow x \equiv y \pmod{n}$$
$$\gcd(a,n) \mid b \Rightarrow \exists x.\ ax \equiv b \pmod{n}$$

Automating divisibility reasoning

Linear (Presburger) arithmetic is a common workhorse in formal verifications. For a lot of the 'congruential' reasoning a custom decision procedure is a similarly useful workhorse:

$$d|a \wedge d|b \Rightarrow d|(a-b)$$
$$\mathrm{coprime}(d,a) \wedge \mathrm{coprime}(d,b) \Rightarrow \mathrm{coprime}(d,ab)$$
$$\mathrm{coprime}(d,ab) \Rightarrow \mathrm{coprime}(d,a)$$
$$\mathrm{coprime}(a,b) \wedge x \equiv y \pmod{a} \wedge x \equiv y \pmod{b} \Rightarrow x \equiv y \pmod{(ab)}$$
$$m|r \wedge n|r \wedge \mathrm{coprime}(m,n) \Rightarrow (mn)|r$$
$$\mathrm{coprime}(xy, x^2+y^2) \Leftrightarrow \mathrm{coprime}(x,y)$$
$$\mathrm{coprime}(a,b) \Rightarrow \exists x.\ x \equiv u \pmod{a} \wedge x \equiv v \pmod{b}$$
$$ax \equiv ay \pmod{n} \wedge \mathrm{coprime}(a,n) \Rightarrow x \equiv y \pmod{n}$$
$$\gcd(a,n) \mid b \Rightarrow \exists x.\ ax \equiv b \pmod{n}$$

For more on how this works, see my paper *Automating elementary number-theoretic proofs using Gröbner bases* (CADE 21):

`https://www.cl.cam.ac.uk/~jrh13/papers/divisibility.pdf`

A common theme: formalized interval arithmetic

In both applications being able to do basic 'interval arithmetic', proving naive or semi-naive bounds on expressions in a formal setting, is very useful.

- On the floating-point side, many simpler arguments just rely on relative error properties of non-denormalizing floating-point roundings, say $\mathtt{round}(x) = x(1 + \epsilon)$ where $|\epsilon| \leq 2^{-53}$, and one just needs to compose them conservatively

A common theme: formalized interval arithmetic

In both applications being able to do basic 'interval arithmetic', proving naive or semi-naive bounds on expressions in a formal setting, is very useful.

- On the floating-point side, many simpler arguments just rely on relative error properties of non-denormalizing floating-point roundings, say $\mathtt{round}(x) = x(1+\epsilon)$ where $|\epsilon| \leq 2^{-53}$, and one just needs to compose them conservatively
- In the crypto bignums, multiplier arrays often appear to throw away carries from computations, and only because you know bounds on those expressions do you know this is safe.

A common theme: formalized interval arithmetic

In both applications being able to do basic 'interval arithmetic', proving naive or semi-naive bounds on expressions in a formal setting, is very useful.

- On the floating-point side, many simpler arguments just rely on relative error properties of non-denormalizing floating-point roundings, say $\mathtt{round}(x) = x(1+\epsilon)$ where $|\epsilon| \leq 2^{-53}$, and one just needs to compose them conservatively
- In the crypto bignums, multiplier arrays often appear to throw away carries from computations, and only because you know bounds on those expressions do you know this is safe.
- Example: if you get a 2-part product $2^{64}h + l = xy$ of two unsigned 64-bit words x and y, you know h can accept an additional 'increment' carry-in without carrying out, because $(2^{64}-1)^2 + 2^{64} < 2^{128}$.

Using Newton's method for division and square root

On the floating-point side, we did lots of verifications of Newton-based algorithms for division and square root. Consider the special case of reciprocals where we want to calculate $1/a$ starting with an approximation y

$$y = \frac{1}{a}(1 + \epsilon)$$

Using Newton's method for division and square root

On the floating-point side, we did lots of verifications of Newton-based algorithms for division and square root. Consider the special case of reciprocals where we want to calculate $1/a$ starting with an approximation y

$$y = \frac{1}{a}(1 + \epsilon)$$

If we then do the following computation

$$e = 1 - a \cdot y$$

we get $e = 1 - a \cdot \frac{1}{a}(1 + \epsilon) = 1 - (1 + \epsilon) = -\epsilon$, ignoring extra rounding errors, so after

Using Newton's method for division and square root

On the floating-point side, we did lots of verifications of Newton-based algorithms for division and square root. Consider the special case of reciprocals where we want to calculate $1/a$ starting with an approximation y

$$y = \frac{1}{a}(1 + \epsilon)$$

If we then do the following computation

$$e = 1 - a \cdot y$$

we get $e = 1 - a \cdot \frac{1}{a}(1 + \epsilon) = 1 - (1 + \epsilon) = -\epsilon$, ignoring extra rounding errors, so after

$$y' = y(1 + e)$$

we get $y' = \frac{1}{a}(1 + \epsilon)(1 - \epsilon) = \frac{1}{a}(1 - \epsilon^2)$, the classic quadratic convergence where we get twice as many bits of accuracy per iteration.

Modular inverses by Hensel lifting

Consider the following requirement for a 1-word (negated) modular inverse

> *Given a 64-bit unsigned and* odd *integer a, find another 64-bit integer x such that $ax \equiv -1 \ (mod\ 2^{64})$, i.e. that* `a * x == 0xFFFFFFFFFFFFFFFF` *using unsigned silently-wrapping word operations like those on C's* `uint64_t`*.*

Modular inverses by Hensel lifting

Consider the following requirement for a 1-word (negated) modular inverse

> *Given a 64-bit unsigned and* odd *integer* a*, find another 64-bit integer* x *such that* $ax \equiv -1 \ (mod\ 2^{64})$*, i.e. that* `a * x == 0xFFFFFFFFFFFFFFFF` *using unsigned silently-wrapping word operations like those on C's* `uint64_t`*.*

This comes up in practice when computing constants for Montgomery multiplication, which is widely used in cryptographic bignums for the combination of efficiency and lack of data-dependent control flow.

Modular inverses by Hensel lifting

Consider the following requirement for a 1-word (negated) modular inverse

> *Given a 64-bit unsigned and* odd *integer* a*, find another 64-bit integer* x *such that* $ax \equiv -1 \ (mod\ 2^{64})$*, i.e. that* `a * x == 0xFFFFFFFFFFFFFFFF` *using unsigned silently-wrapping word operations like those on C's* `uint64_t`*.*

This comes up in practice when computing constants for Montgomery multiplication, which is widely used in cryptographic bignums for the combination of efficiency and lack of data-dependent control flow.

It can be solved in a directly similar way using *Hensel lifting*, the p-adic analog of Newton's method.

Initial approximation

As with the floating-point inverse, we need an initial approximation to start with. The following piece of magic (in C syntax, carat = XOR)

```
x = (a - (a<<2))^2
```

just so happens to give a 5-bit negated modular inverse, i.e. a value with $ax \equiv -1 \pmod{2^5}$ (assuming we start with an odd a, of course).

Hensel lifting step

Suppose we have a k-bit approximation $ax \equiv -1 \pmod{2^k}$ and we do the same sort of Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

Hensel lifting step

Suppose we have a k-bit approximation $ax \equiv -1 \pmod{2^k}$ and we do the same sort of Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

By the initial assumption $ax = 2^k n - 1$ for some integer n

Hensel lifting step

Suppose we have a k-bit approximation $ax \equiv -1 \pmod{2^k}$ and we do the same sort of Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

By the initial assumption $ax = 2^k n - 1$ for some integer n
So $e = ax + 1 = 2^k n$ and $e + 1 = 2^k n + 1$ and
$ay = ax(e + 1) = (2^k n - 1)(2^k n + 1) = 2^{2k} n^2 - 1$, i.e.
$ay \equiv -1 \pmod{2^{2k}}$.

Hensel lifting step

Suppose we have a k-bit approximation $ax \equiv -1 \pmod{2^k}$ and we do the same sort of Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

By the initial assumption $ax = 2^k n - 1$ for some integer n
So $e = ax + 1 = 2^k n$ and $e + 1 = 2^k n + 1$ and
$ay = ax(e + 1) = (2^k n - 1)(2^k n + 1) = 2^{2k} n^2 - 1$, i.e.
$ay \equiv -1 \pmod{2^{2k}}$.
These will in practice be done with word operations mod 2^{64}, so actually the congruence holds mod $2^{\min(2k,64)}$. We can repeat, doubling k each time till we max out at the word size.

The Goldschmidt variant

There are elaborations of Newton's method where we compute the next error in parallel, which in this case happens to go from ϵ to ϵ^2, so this is an alternative to give the full functionality:

```
x = (a - (a<<2))^2;
e = a * x + 1;
x = e * x + x; e = e * e;
x = e * x + x; e = e * e;
x = e * x + x; e = e * e;
x = e * x + x;
```

The Goldschmidt variant

There are elaborations of Newton's method where we compute the next error in parallel, which in this case happens to go from ϵ to ϵ^2, so this is an alternative to give the full functionality:

```
x = (a - (a<<2))^2;
e = a * x + 1;
x = e * x + x; e = e * e;
x = e * x + x; e = e * e;
x = e * x + x; e = e * e;
x = e * x + x;
```

Yet another variant is to compute a power series in the initial error and use that directly to compute a more accurate answer.

The Goldschmidt variant

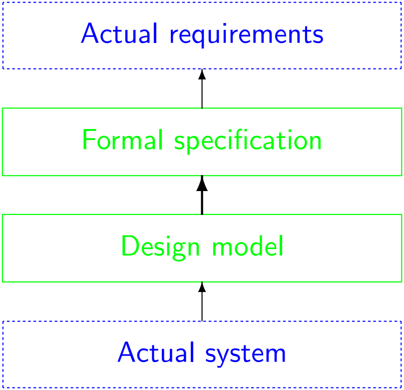
There are elaborations of Newton's method where we compute the next error in parallel, which in this case happens to go from ϵ to ϵ^2, so this is an alternative to give the full functionality:

```
x = (a - (a<<2))^2;
e = a * x + 1;
x = e * x + x; e = e * e;
x = e * x + x; e = e * e;
x = e * x + x; e = e * e;
x = e * x + x;
```

Yet another variant is to compute a power series in the initial error and use that directly to compute a more accurate answer.
In the floating-point setting these variants lose the 'self-correcting' property of Newton iteration and so need care in later iterations.
In the integer setting everything holds modulo the wordsize and we don't need to worry!

Modeling the system, formalizing the spec

One nice thing about formalizing arithmetic (versus many other settings) is that the spec is pretty easy to formalize, almost formal already.



Modeling the system is more challenging.

Machine code modeling overview

Our current crypto verifications are for x86_64 and ARM8 machine code, using a simple relational model of the execution, e.g.

```
|- x86_ADD dest src s =
        let x = read dest s and y = read src s in
        let z = word_add x y in
        (dest := (z:N word) ,,
         ZF := (val z = 0) ,,
         SF := (ival z < &0) ,,
         PF := word_evenparity(word_zx z:byte) ,,
         CF := ~(val x + val y = val z) ,,
         OF := ~(ival x + ival y = ival z) ,,
         AF := ~(val(word_zx x:nybble) +
                 val(word_zx y:nybble) =
                 val(word_zx z:nybble))) s
```

Most examples like this are deterministic, but the model has some nondeterminism (e.g. some flags are undefined according to the ISA).

Verification flow

Our approach is to verify pre-existing machine code, not autogenerate 'correct by construction' code (but we usually write the code ourselves).

Verification flow

Our approach is to verify pre-existing machine code, not autogenerate 'correct by construction' code (but we usually write the code ourselves).

- ☺ Independent of compiler or even macro-assembler correctness.

Verification flow

Our approach is to verify pre-existing machine code, not autogenerate ‘correct by construction’ code (but we usually write the code ourselves).

- ☺ Independent of compiler or even macro-assembler correctness.
- ☺ Applicable to highly tuned efficient code that is hard to generate automatically as well as compiled C code etc.

Verification flow

Our approach is to verify pre-existing machine code, not autogenerate 'correct by construction' code (but we usually write the code ourselves).

- ☺ Independent of compiler or even macro-assembler correctness.
- ☺ Applicable to highly tuned efficient code that is hard to generate automatically as well as compiled C code etc.
- ☺ Code is conventional human-readable and human-modifiable, usage is independent of prover infrastructure.

Verification flow

Our approach is to verify pre-existing machine code, not autogenerate ‘correct by construction’ code (but we usually write the code ourselves).

- ☺ Independent of compiler or even macro-assembler correctness.
- ☺ Applicable to highly tuned efficient code that is hard to generate automatically as well as compiled C code etc.
- ☺ Code is conventional human-readable and human-modifiable, usage is independent of prover infrastructure.
- ☹ Much more work involved writing code at this level, less structured representation.

Verification flow

Our approach is to verify pre-existing machine code, not autogenerate ‘correct by construction’ code (but we usually write the code ourselves).

- ☺ Independent of compiler or even macro-assembler correctness.
- ☺ Applicable to highly tuned efficient code that is hard to generate automatically as well as compiled C code etc.
- ☺ Code is conventional human-readable and human-modifiable, usage is independent of prover infrastructure.
- ☹ Much more work involved writing code at this level, less structured representation.
- ☺/☹ Exposure of low-level details like exact stack and PC offsets and particular registers.

Verification results

The final verification results take the form of elaborated Hoare triples where as well as the precondition and postcondition there is a ‘frame condition’ asserting which parts of the state may change:

```
|- ODD(val a)
   ==> ensures arm
        (\s. aligned_bytes_loaded s (word pc) word_negmodinv_mc /\
             read PC s = word pc /\
             read X0 s = a)
        (\s'. read PC s' = word (pc + 48) /\
              (val a * val(read X0 s') + 1 == 0) (mod (2 EXP 64)))
        (MAYCHANGE [PC; X0; X1; X2])
```

Verification results

The final verification results take the form of elaborated Hoare triples where as well as the precondition and postcondition there is a ‘frame condition’ asserting which parts of the state may change:

```
|- ODD(val a)
   ==> ensures arm
        (\s. aligned_bytes_loaded s (word pc) word_negmodinv_mc /\
             read PC s = word pc /\
             read X0 s = a)
        (\s'. read PC s' = word (pc + 48) /\
              (val a * val(read X0 s') + 1 == 0) (mod (2 EXP 64)))
        (MAYCHANGE [PC; X0; X1; X2])
```

The third field makes it reasonably straightforward to compose results, e.g. for function calls, repeatedly inlined sections of code.

No separation logic

We don't use any formalization of separation logic but state and reason about overlap directly using that third field.

No separation logic

We don't use any formalization of separation logic but state and reason about overlap directly using that third field.

- ☹ Involves writing explicit non-aliasing hypotheses on theorems and reasoning about them

No separation logic

We don't use any formalization of separation logic but state and reason about overlap directly using that third field.

- ☹ Involves writing explicit non-aliasing hypotheses on theorems and reasoning about them
- ☺ Most reasoning is automated, keeps specifications explicit and flexible.

Example of explicit nonaliasing

A spec for a $6 \times 6 \to 12$ word multiplier (core without register save/restore and return).

```
 |- nonoverlapping (word pc,0x2e4) (z,8 * 12) /\
    (y = z \/ nonoverlapping (y,8 * 6) (z,8 * 12)) /\
    nonoverlapping (x,8 * 6) (z,8 * 12)
     ==> ensures x86
          (\s. bytes_loaded s (word pc) bignum_mulx_6_12_mc /\
               read RIP s = word(pc + 0x06) /\
               C_ARGUMENTS [z; x; y] s /\
               bignum_from_memory (x,6) s = a /\
               bignum_from_memory (y,6) s = b)
          (\s. read RIP s = word (pc + 0x2dd) /\
               bignum_from_memory (z,12) s = a * b)
          (MAYCHANGE [RIP; RAX; RBP; RBX; RCX; RDX;
                      R8; R9; R10; R11; R12; R13] ,,
           MAYCHANGE [memory :> bytes(z,8 * 12)] ,,
           MAYCHANGE SOME_FLAGS)
```

Note that the second input argument y can be the same as the output buffer z.

The verification process

The approach we've settled on is a combination of Floyd-Hoare rules for the top-level decomposition (loop invariants, sequential composition) and then mainly symbolic simulation by proof below that.

The verification process

The approach we've settled on is a combination of Floyd-Hoare rules for the top-level decomposition (loop invariants, sequential composition) and then mainly symbolic simulation by proof below that.

For instance the negated modular inverse proof naturally divides into two pieces, one for the initial approximation (proved by bit-blasting case splits) and one for the Hensel lifting (proved by congruence reasoning), and we use this step to break the proof into two with the intermediate spec:

```
ENSURES_SEQUENCE_TAC `pc + 12`
  `\s. (a * val (read X1 s) + 1 == 0) (mod 16) /\
       read X0 s = word a`
```

Continuous integration

To minimize the chance of a ‘gap at the bottom’ between the code being verified and the code being run, we directly generate the verification target from the object file, recording what is expected as a sanity check:

Continuous integration

To minimize the chance of a ‘gap at the bottom’ between the code being verified and the code being run, we directly generate the verification target from the object file, recording what is expected as a sanity check:

```
let word_negmodinv_mc =
define_assert_from_elf "word_negmodinv_mc" "Arm/wordnegmodinv.o"
[ 0xd37ef401;       (* arm_LSL X1 X0 (rvalue (word 2)) *)
  0xcb010001;       (* arm_SUB X1 X0 X1 *)
  0xd27f0021;       (* arm_EOR X1 X1 (rvalue (word 2)) *)
  0xd2800022;       (* arm_MOV X2 (rvalue (word 1)) *)
  0x9b010802;       (* arm_MADD X2 X0 X1 X2 *)
  0x9b027c40;       (* arm_MUL X0 X2 X2 *)
  0x9b010441;       (* arm_MADD X1 X2 X1 X1 *)
  0x9b007c02;       (* arm_MUL X2 X0 X0 *)
  0x9b010401;       (* arm_MADD X1 X0 X1 X1 *)
  0x9b027c40;       (* arm_MUL X0 X2 X2 *)
  0x9b010441;       (* arm_MADD X1 X2 X1 X1 *)
  0x9b010400;       (* arm_MADD X0 X0 X1 X1 *)
  0xd65f03c0        (* arm_RET X30 *)
];;
```

The right-hand column is autogenerated and consists only of `objdump`-style comments for documentation.

In conclusion

- Theorem provers like HOL Light, ACL2 and others are indeed general-purpose and can be applied to all levels of diverse verification tasks

In conclusion

- Theorem provers like HOL Light, ACL2 and others are indeed general-purpose and can be applied to all levels of diverse verification tasks
- In some ways the reals and the integers bring up very different problems, but there are many interesting common themes and analogies between the two worlds

In conclusion

- Theorem provers like HOL Light, ACL2 and others are indeed general-purpose and can be applied to all levels of diverse verification tasks
- In some ways the reals and the integers bring up very different problems, but there are many interesting common themes and analogies between the two worlds
- Programmability of a proof assistant is a tremendous boon since these verification challenges often require specialized inference rules not matching off-the-shelf solvers.