Verifying Algorithms for the Transcendental Functions

John Harrison

University of Cambridge

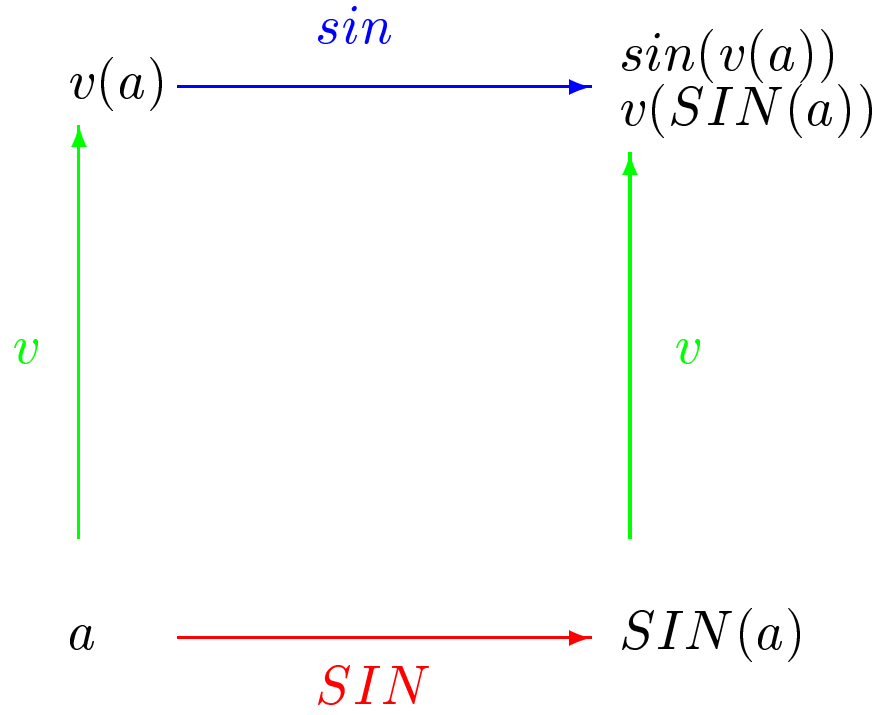
- Introduction and motivation
- Floating point correctness
- Quick summary of HOL Light
- Our programming language
- Example 1: CORDIC method for logarithm
- Example 2: polynomial for exponential

Introduction and motivation

- Floating point algorithms are fairly small, but often complicated mathematically.
- There have been errors in commercial systems, e.g. the Pentium FDIV bug in 1994.
- In the case of transcendental functions it's difficult even to say what correctness means.
- Verification using model checkers is difficult because of the need for mathematical apparatus.
- It can even be difficult using theorem provers since not many of them have good theories of real numbers etc.

Floating point correctness (1)

We want to specify the correctness according to the following diagram:



What relationship between $v(SIN(a))$ and $sin(v(a))$ should we require?

Floating point correctness (2)

There are various plausible options, all of which are easy to express formally in HOL:

- The answer is the closest representable number to the true answer (with round to even in case of two equally close answers)
- The absolute error is small.
- The relative error is small.
- The error is commensurate with the likely error in the input.

All of these can be modified 'probabilistically', e.g. one can say that the answer is the closest to the true answer in 99.9% of cases.

Quick summary of HOL Light

- LCF-style theorem prover based on classical higher order logic (simple type theory).
- The LCF approach makes it programmable and reliable, though sometimes a bit slow.
- Fairly extensive mathematical infrastructure including real analysis.
- It has evolved via:
  - Edinburgh LCF (Milner et al.)
  - Cambridge LCF (Paulson)
  - HOL (Gordon, Melham)
  - hol90 (Slind)

  Other LCF-style systems include Nuprl, Coq and Isabelle.

The LCF approach

The key ideas are:

- All theorems created by low-level primitive rules.
- Guaranteed by using an abstract type of theorems; no need to store proofs.
- ML available for implementing derived rules by arbitrary programming.

This gives advantages of reliability and extensibility. The system's source code can be completely open. The user controls the means of production (of theorems). To improve efficiency one can:

- Encapsulate reasoning in single theorems.
- Separate proof search and proof checking.

Some of HOL Light's derived rules

- Simplifier for (conditional, contextual) rewriting.
- Tactic mechanism for mixed forward and backward proofs.
- Tautology checker.
- Automated theorem provers for pure logic, based on tableaux and model elimination.
- Tools for definition of (infinitary, mutually) inductive relations.
- Tools for definition of (mutually) recursive datatypes
- Linear arithmetic decision procedures over $\mathbb{R}$, $\mathbb{Z}$ and $\mathbb{N}$.
- Differentiator for real functions.

Real analysis theory (1)

- Definitional construction of real numbers
- Basic topology
- General limit operations
- Sequences and series
- Limits of real functions
- Differentiation
- Power series and Taylor expansions
- Transcendental functions
- Gauge integration

Real analysis theory (2)

There are lots of concrete theorems, e.g.

```
|- abs(abs x - abs y) <= abs (x - y)

|- sin(x + y) =
   sin(x) * cos(y) + cos(x) * sin(y)

|- tan(&n * pi) = &0

|- &0 < x /\ &0 < y
   ==> (ln(x / y) = ln(x) - ln(y))
```

Real analysis theory (3)

and many general ones:

```
|- f contl x /\ g contl (f x)
   ==> (\x. g(f x)) contl x

|- a <= b /\
   (f(a) <= y /\ y <= f(b)) /\
   (!x. a <= x /\ x <= b ==> f contl x)
   ==> (?x. a <= x /\ x <= b /\ (f(x) = y))

|- (f diffl l)(g x) /\ (g diffl m)(x)
   ==> ((\x. f(g x)) diffl (l * m))(x)

|- a <= b /\
   (!x. a <= x /\ x <= b
        ==> (f diffl f'(x))(x))
   ==> Dint(a,b) f' (f(b) - f(a))
```

Our Programming Language (1)

This includes the following constructs:

command = `variable :=` *expression*
| *command* `;` *command*
| `if` *expression* `then` *command*
`else` *command*
| `if` *expression* `then` *command*
| `while` *expression* `do` *command*
| `do` *command* `while` *expression*
| `skip`
| { *expression*}
| [*expression*]

The language is semantically embedded in HOL using standard techniques (functional, relational and weakest precondition semantics are all available).

Our Programming Language (2)

We can verify the total correctness of programs according to given pre and post-conditions.

```
|- correct p c q
```

corresponds to the standard total correctness assertion $[p]\ c\ [q]$, i.e. a command c, executed in a state satisfying p, will terminate in a state satisfying q.

We can prove correctness assertions by systematically breaking down the command according to its structure. In particular, we can annotate it with 'verification conditions', and so (automatically) reduce the correctness proof to the problem of verifying some assertions about the underlying mathematical domains.

Example 1: CORDIC for logarithm

```
begin
  var k,x,y,z;
  x := X;
  y := 0;
  k := 1;
  while k < N do
   ( z := srl(n) k x;
     if ult(n) z (neg(n) x) then
       (x := add(n) x z;
        y := add(m) y (logs k));
     k := k + 1
   )
end
```

where `add(n)`, `neg(n)`, `ult(n)` and `srl(n) k` are n-bit addition, 2s complement negation, unsigned comparison ($<$) and right shift by k places, respectively.

The array `logs` contains pre-stored constants.

Without the prettyprinter

This shows what the underlying semantic representation looks like:

```
Assign (\k,(x,(y,z)). k,(X,(y,z))) Seq
Assign (\k,(x,(y,z)). k,(x,(0,z))) Seq
Assign (\k,(x,(y,z)). 1,(x,(y,z))) Seq
While (\k,(x,(y,z)). k < N)
(Assign (\k,(x,(y,z)).
             k,(x,(y,srl n k x))) Seq
 If (\k,(x,(y,z)). ult n z (neg n x))
 (Assign (\k,(x,(y,z)).
              k,(add n x z,(y,z))) Seq
  Assign
     (\k,(x,(y,z)).
        k,(x,(add m y (logs k),z)))) Seq
 Assign (\k,(x,(y,z)). k + 1,(x,(y,z))))
```

However the user need not normally see this form!

The CORDIC program in C

```
int k;
unsigned long x,y,z;
x = X;
y = 0;
k = 1;
while (k < N)
 { z = x >> k;
   if (z < -x)
    { x = x + z;
      y = y + logs[k];
    }
   k = k + 1;
 }
```

(Using unsigned longs in place of the particular word sizes, for the sake of familiarity.)

The CORDIC program in Verilog

```
integer k;
reg [n:0] x,z;
reg [m:0] y;
initial;
begin
  x = X;
  y = 0;
  k = 1;
  while (k < N)
  begin
     z = x >> k;
     if (z < -x)
     begin
       x = x + z;
       y = y + logs[k];
     end
     k = k + 1;
  end
end
```

Annotations for CORDIC program

We can specify intermediate assertions later in the proof by exploiting metavariables. However it is simpler to provide annotations. We assert a loop invariant:

```
{mval(n) x < &1 /\ ...}
```

and that `N - k` decreases with each iteration.

The automatic verification condition generator (working by inference) can calculate all the other intermediate assertions for itself. We are left with four verification conditions:

- The loop invariant is true initially.
- The loop invariant is preserved if the condition in the `if` statement holds.
- The loop invariant is preserved if the condition in the `if` statement does not hold.
- The loop invariant together with $k \geq N$ implies the final postcondition.

Correctness result (1)

The four verification conditions are proved in HOL, with the aid of a few lemmas. This proves that the annotated program is correct according to the specification. HOL Light then proves automatically that the program with the annotations removed is still correct. The precondition of the final specification is:

```
inv(&2) <= mval(n) X /\ mval(n) X < &1 /\
&N + &2 <= &n /\ &N <= &2 pow (PRE n) /\
(!i. &0 < &i /\ &i < &N ==>
     &2 pow i * &(logs i) <= &2 pow m /\
     (abs(&(logs i) -
      &2 pow m * ln(&1 + inv(&2 pow i)))
      < &1))
```

i.e. the input value X is in the range $\frac{1}{2} \leq X < 1$, the stored constants are good enough approximations to the true logarithms, and a few conditions on the parameters hold.

Correctness result (2)

and the final postcondition guaranteed by our proof is:

```
abs(mval(m) y + ln(mval(n) X))
    <= &N * (&6 * inv(&2 pow n) +
             inv(&2 pow m)) +
        inv(&2 pow N)
```

That is, the difference between the calculated logarithm `mval(m) y` and (the negation of) the true mathematical result `ln(mval(n) X)` is bounded by $N(6.2^{-n} + 2^{-m}) + 2^{-N}$.

This can be chosen as small as desired by picking the parameters appropriately. Moreover the correct values for the stored table of logarithms can also be calculated in any particular instant, by inference (slowly!)

Example 2: Polynomial for exp

Many modern algorithms for the transcendental functions work according to the following scheme. To calculate $f(x)$:

- We have a pre-stored $f(a_i)$ for some a_i close to x. Consider the difference $x' = x - a_i$.
- Evaluate some similar function $g(x')$. Since x' is small, this can be done accurately by a low-order polynomial approximation $p(x')$.
- Reconstruct $f(x)$ from $f(a_i)$ and $g(x')$

For example, we can calculate $e^x = e^{a_i} e^{x'}$. Errors appear in several places. The hardest to quantify, in a theorem prover, is simply the difference between $p(x')$ and $g(x')$. We focus on this problem here.

Minimax approximations

Polynomial approximations are normally chosen to have the best 'minimax' behaviour, i.e. to minimize the maximum value of the absolute error over the interval $[a, b]$ concerned:

$$||f(x) - p(x)||_\infty = sup_{a \leq x \leq b}|f(x) - p(x)|$$

By a theorem of Chebyshev, such a polynomial always exists. However there is no (known) method for obtaining the coefficients 'analytically'.

Just truncating the Taylor series usually gives a much worse result.

Instead, the coefficients of p are usually arrived at via a rather complicated method involving successive numerical approximations.

It's quite impractical to do all this inside a theorem prover.

Post-hoc error bounds

Instead, we accept the polynomial chosen, and try to find the maximum error $|e(x)|$ directly, where $e(x) = f(x) - p(x)$.

Since the function $f(x)$ is normally well-behaved over the small interval concerned, we can arrive at a pessimistic uniform continuity bound B such that:

$$\forall x, x' \in [a, b].\ |e(x) - e(x')| \leq B|x - x'|$$

Now in principle we can find the upper and lower bounds to accuracy ϵ just by evaluating $e(x)$ at a family of points at most ϵ/B apart and finding the maximum and minimum.

In practice, the number of calculations this involves makes it completely impractical, at least inside a theorem prover.

Using the derivative

A better approach would be to find all the points where $e'(x) = 0$.

We already have a theorem in HOL asserting that if a function is differentiable in an interval, its extrema must occur either at the endpoints or at a point of zero derivative.

Thus, all we need to do is locate all the roots of $e'(x)$ in the interval to accuracy ϵ/B and evaluate $e(x)$ there.

If a root x is simple, then we can give $\alpha < x < \beta$ such that $e'(\alpha)$ and $e'(\beta)$ have opposite signs. The existence of a root in the interval then follows and can be proved in HOL.

But how do we know that we have located *all* the roots? We need this to apply the main theorem. There seems no easy way of proving that.

Using polynomials

Instead, following a suggestion of David Wheeler, we take the following approach.

Approximate $f(x)$ by a truncated Taylor expansion $t(x)$.

Make the degree of $t(x)$ much higher than the polynomial we are concerned with, if necessary, in order to make the error in approximation less than $\epsilon/2$.

Now if suffices to find the maximum and minimum of $e(x) = t(x) - p(x)$ to within $\epsilon/2$.

This is a tractable problem. We can decide, using Sturm's theorem, how many roots $e'(x)$ has in the interval, and so prove that we have located all of them.

Thus we can find the extrema. Moreover, we only need to do rational arithmetic.

Polynomials in HOL

We first develop a theory of polynomials in HOL.

```
|- (poly [] x = &0) /\
   (poly (CONS h t) x = h + x * poly t x)
```

We can define various operations on the coefficients. For example, these correspond to addition and differentiation:

```
|- ([] ++ l2 = l2) /\
   (CONS h t ++ l2 =
     (l2 = []) => CONS h t |
         CONS (h + (HD l2)) (t ++ (TL l2)))

|- (diff_aux n [] = []) /\
   (diff_aux n (CONS h t) =
      CONS (&n * h) (diff_aux (SUC n) t))

|- diff l =
      ((l = []) => [] | diff_aux 1 (TL l))
```

Correctness of operations

These operations do indeed work as intended:

```
|- !p1 p2 x. poly (p1 ++ p2) x =
            poly p1 x + poly p2 x

|- !p c x. poly (c ## p) x = c * poly p x

|- !p x. poly (neg p) x = --(poly p x)

|- !x p1 p2. poly (p1 ** p2) x =
            poly p1 x * poly p2 x

|- !l x. ((poly l) diffl
          (poly (diff l) x)) x
```

Hence we get various basic properties of the operations. Note that not all of them hold at the level of coefficient lists, which are not canonical.

Theory of polynomials

Now we define other basic notions and prove various important theorems, e.g.

- If $p(a) = 0$ then $p(x)$ is divisible by $(x - a)$
- If $(x - a)$ divides $p(x)q(x)$ then $(x - a)$ divides $p(x)$ or $(x - a)$ divides $q(x)$.
- A nontrivial polynomial only has finitely many roots.
- For a nontrivial polynomial p, every a is characterized by a unique order such that $(x - a)^n$ divides p but $(x - a)^{n+1}$ doesn't.

For example in HOL:

```
|- !p. ~(poly p = poly [])
        ==> FINITE {x | poly p x = &0}
```

Squarefree decomposition (1)

When trying to locate the zeros of a polynomial $p(x)$, it's useful to assume that it has no multiple roots, i.e. every root has order 1.

In that case, we can rely on $p(\alpha)$ and $p(\beta)$ having opposite signs for a small enough isolating interval, and so prove in HOL that the root exists.

Moreover, Sturm's theorem is easier to prove assuming that the polynomial concerned has no multiple (real) roots.

```
|- !p. rsquarefree p =
        ~(poly p = poly []) /\
        (!a. (order a p = 0) \/
             (order a p = 1))
```

This says that a polynomial has only simple real roots.

Squarefree decomposition (2)

We prove that this is equivalent to the polynomial and its derivative having no common roots:

```
|- !p. rsquarefree p =
        (!a. ~((poly p a = &0) /\
               (poly (diff p) a = &0)))
```

From this, it follows that $p/gcd(p, p')$ gives a polynomial with the same roots but all of order 1. This is the *squarefree decomposition* of p.

Rather than formalize division, gcd etc. we calculate appropriate polynomials outside the logic (using Maple) which are sufficient as a certificate that d is a g.c.d.

The appropriate constants in a Bezout identity are given by Maple's `gcdex` functions.

Squarefree decomposition (3)

The final HOL theorem concerning squarefree decomposition is:

```
|- !p q d e r s.
     ~(poly (diff p) = poly []) /\
     (poly p = poly (q ** d)) /\
     (poly (diff p) = poly (e ** d)) /\
     (poly d = poly (r ** p ++ s ** diff p))
     ==> rsquarefree q /\
         (!a. (poly q a = &0) =
              (poly p a = &0))
```

Now we only need concern ourselves with counting and locating the roots of a squarefree polynomial.

Sturm's theorem (1)

In order to count the roots, we use the notion of a *Sturm sequence*.

```
|- (STURM p p' [] = p' divides p) /\
   (STURM p p' (CONS g gs) =
      p' divides (p ++ g) /\
      degree g < degree p' /\
      STURM p' g gs)
```

The standard Sturm sequence starts with p' the derivative of the original polynomial p. By successive division (and negation) we get a Sturm sequence.

The number of roots of p in an interval can be found by counting the variations in sign of the Sturm sequence evaluated at each end of the interval.

Sturm's theorem (2)

We define the number of variations in sign of a sequence of real numbers (ignoring zeros) as follows:

```
|- (varrec prev [] = 0) /\
   (varrec prev (CONS h t) =
      (prev * h < &0
       => SUC (varrec h t)
        | h = &0 => varrec prev t
                  | varrec h t))

|- variation l = varrec (&0) l
```

Sturm's theorem asserts that if a polynomial p is nonzero at either end of an interval $[a, b]$, then the number of roots it has inside the interval is the difference in the variations of the Sturm sequence at the end points.

Sturm's theorem (3)

```
|- !f a b l.
       a <= b /\
       ~(poly f a = &0) /\
       ~(poly f b = &0) /\
       rsquarefree f /\
       STURM f (diff f) l
       ==> {x | a <= x /\ x <= b /\
                 (poly f x = &0)} HAS_SIZE
           (variation
             (MAP (\p. poly p a)
               (CONS f (CONS (diff f) l))) -
            variation
             (MAP (\p. poly p b)
               (CONS f (CONS (diff f) l))))
```