Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 11

ML examples III: Exact Real Arithmetic

Topics covered:

- Finite representations
- Real numbers as approximating functions
- Arbitrary precision integers
- Operations over reals
- Caching

Finite representations

Real arithmetic on computers is normally done via floating point approximations.

In general, we can only manipulate a real number, either ourselves or inside a computer, via some sort of finite representation.

Some question how numbers can be said to 'exist' if they have no finite representation.

For example, Kronecker accepted integers and rationals because they can be written down explicitly, and even *algebraic* numbers because they can be represented using the polynomials of which they are solutions.

However he rejected transcendental numbers because apparently they could not be represented finitely.

Real numbers as programs

However, given our modern perspective, we can say that after all many more numbers than Kronecker would have accepted *do* have a finite representation.

This is the program used to calculate them to greater and greater precision.

For example, we can write a program that will produce for any argument n the first n digits of π.

Alternatively it can produce a rational number r such that $|\pi - r| < 2^{-n}$.

Whatever approach is taken to the successive approximation of a real number, the key point is that its representation, the program itself, is finite.

Our representation of reals

We represent a real x by a function $f_x : \mathbb{N} \to \mathbb{Z}$ that for each $n \in \mathbb{N}$:

$$|f_x(n) - 2^n x| < 1$$

This is of course equivalent to

$$|\frac{f_x(n)}{2^n} - x| < \frac{1}{2^n}$$

We can actually represent the arithmetic operations on numbers as higher order functions.

Given functions for approximating x and y, will produce new ones for approximating $x + y$, xy, $sin(x)$ and so on, for a wide range of functions.

Such a result is exact, in the sense that we can then give it an arbitrary desired precision and it will perform the appropriate calculation automatically.

Arbitrary precision integers

CAML's standard integers (type `int`) have a severely limited range, so first of all we need to set up a type of unlimited-precision integers.

The CAML release includes a library that gives a fast implementation of arbitrary precision integer (and in fact rational) arithmetic.

A version of CAML Light with this library pre-loaded is installed on Thor. Just run it with:

```
$ camllight my_little_caml
```

then do the following to get access to all the functions:

```
#open "num";;
```

This library sets up a new type `num` of arbitrary precision rational numbers; we will just use the integer subset.

Operators on `num`

CAML does not provide overloading, so it is necessary to use different symbols for the usual arithmetic operations over `num`.

Note that small constants of type `num` must be written as `Int k` rather than simply `k`.

The unary negation on type `num` is written `minus_num`. Another handy unary operator is `abs_num`, which finds absolute values, i.e. given x returns $|x|$.

The usual complement of binary operations are also available. The (truncating) division and modulus function, called `quo_num` and `mod_num`, do not have infix status. However most of the other binary operators are infixes, and the names are derived from the usual ones by adding a slash '`/`'.

Binary operators

Op	Type	Meaning
`**/`	`num -> num -> num`	Exponentiation
`*/`	`num -> num -> num`	Multiplication
`+/`	`num -> num -> num`	Addition
`-/`	`num -> num -> num`	Subtraction
`=/`	`num -> num -> bool`	Equality
`<>/`	`num -> num -> bool`	Inequality
`</`	`num -> num -> bool`	Less than
`<=/`	`num -> num -> bool`	Less or equal
`>/`	`num -> num -> bool`	Greater than
`>=/`	`num -> num -> bool`	Greater or equal

Getting started

Recall that our real numbers are supposed to be (represented by) functions $\mathbb{Z} \to \mathbb{Z}$.

In ML we will actually use `int -> num`, since the inbuilt type of integers is more than adequate for indexing the level of accuracy.

Now we can define some basic operations on reals. The most basic operation, which gets us started, is to produce the real number corresponding to an integer. This is easy:

```
#let real_of_int k n =
   (Int 2 **/ Int n) */ Int k;;
real_of_int : int -> int -> num = <fun>
#real_of_int 23;;
- : int -> num = <fun>
```

Evidently this satisfies the error criterion: in fact the error is zero.

Basic operations

Now we can define the first nontrivial operation, that of unary negation:

```
let real_neg f n = minus_num(f n);;
```

The compiler generalizes the type more than intended, but this will not trouble us. It is almost as easy to see that the approximation criterion is preserved. If we know that for each n:

$$|f_x(n) - 2^n x| < 1$$

then we have for any n:

$$\begin{aligned} |f_{-x}(n) - 2^n(-x)| &= |-f_x(n) - 2^n(-x)| \\ &= |-(f_x(n) - 2^n x)| \\ &= |f_x(n) - 2^n x| \\ &< 1 \end{aligned}$$

Similarly, we can define an 'absolute value' function on real numbers, using the corresponding function `abs_num` on numbers.

Addition: first attempt

We could define:

$$f_{x+y}(n) = f_x(n) + f_y(n)$$

However this gives no guarantee that the approximation criterion is maintained; we would have:

$$\begin{aligned}
& |f_{x+y}(n) - 2^n(x+y)| \\
= \quad & |f_x(n) + f_y(n) - 2^n(x+y)| \\
\leq \quad & |f_x(n) - 2^n x| + |f_y(n) - 2^n y|
\end{aligned}$$

We can guarantee that the sum on the right is less than 2, but not that it is less than 1 as required. Therefore, we need in this case to evaluate x and y to *greater* accuracy than required in the answer.

Addition: second attempt

Suppose we define:

$$f_{x+y}(n) = (f_x(n+1) + f_y(n+1))/2$$

Now we have:

$$\begin{aligned}
& |f_{x+y}(n) - 2^n(x+y)| \\
= \quad & |(f_x(n+1) + f_y(n+1))/2 - 2^n(x+y)| \\
\leq \quad & |f_x(n+1)/2 - 2^n x| + |f_y(n+1)/2 - 2^n y| \\
= \quad & \frac{1}{2}|f_x(n+1) - 2^{n+1}x| + \frac{1}{2}|f_y(n+1) - 2^{n+1}y| \\
< \quad & \frac{1}{2}1 + \frac{1}{2}1 = 1
\end{aligned}$$

Apparently this just gives the accuracy required. However we have implicitly used real mathematical division above. Since the function is supposed to yield an integer, we are obliged to round the quotient to an integer.

Rounding division

If we just use quo_num, the error from rounding this might be almost 1, after which we could never guarantee the bound we want, however accurately we evaluate the arguments.

We need a division function that always returns the integer closest to the true result (or one of them in the case of two equally close ones), so that the rounding error never exceeds $\frac{1}{2}$.

```
#let ndiv x y = round_num(x // y);;
ndiv : num -> num -> num = <fun>
##infix "ndiv";;
#(Int 23) ndiv (Int 5);;
- : num = Int 5
#(Int 22) ndiv (Int 5);;
- : num = Int 4
```

Now we are ready to define a correct addition function!

Addition: third attempt

Now if we define:

$$f_{x+y}(n) = (f_x(n+2) + f_y(n+2)) \text{ ndiv } 4$$

everything works:

$$\begin{aligned}
& |f_{x+y}(n) - 2^n(x+y)| \\
= \;& |((f_x(n+2) + f_y(n+2)) \text{ ndiv } 4) - 2^n(x+y)| \\
\leq \;& \frac{1}{2} + |(f_x(n+2) + f_y(n+2))/4 - 2^n(x+y)| \\
= \;& \frac{1}{2} + \frac{1}{4}|(f_x(n+2) + f_y(n+2)) - 2^{n+2}(x+y)| \\
\leq \;& \frac{1}{2} + \frac{1}{4}|f_x(n+2) - 2^{n+2}x| + \\
& \frac{1}{4}|f_y(n+2) - 2^{n+2}y| \\
< \;& \frac{1}{2} + \frac{1}{4}1 + \frac{1}{4}1 = 1
\end{aligned}$$

Accordingly we make our definition:

```
let real_add f g n =
  (f(n + 2) +/ g(n + 2)) ndiv (Int 4);;
```

Multiplication by an integer (1)

It's worth treating this special case efficiently. We define:

$$f_{mx}(n) = (m f_x(n+p+1)) \text{ ndiv } 2^{p+1}$$

where p is chosen so that $2^p \geq |m|$. For correctness, we have:

$$\begin{array}{rl} & |f_{mx}(n) - 2^n(mx)| \\ \leq & \frac{1}{2} + |\frac{m f_x(n+p+1)}{2^{p+1}} - 2^n(mx)| \\ = & \frac{1}{2} + \frac{|m|}{2^{p+1}} |f_x(n+p+1) - 2^{n+p+1}x| \\ < & \frac{1}{2} + \frac{|m|}{2^{p+1}} \\ \leq & \frac{1}{2} + \frac{1}{2}\frac{|m|}{2^p} \\ \leq & \frac{1}{2} + \frac{1}{2} = 1 \end{array}$$

Multiplication by an integer (2)

In order to implement this, we need a function to find the appropriate p:

```
let log2 =
  let rec log2 x y =
    if x </ Int 1 then y
    else log2 (quo_num x (Int 2)) (y + 1) in
  fun x -> log2 (x -/ Int 1) 0;;
```

The implementation is simply:

```
let real_intmul m x n =
  let p = log2 (abs_num m) in
  let p1 = p + 1 in
  (m */ x(n + p1)) ndiv (Int 2 **/ Int p1);;
```

For division by an integer, we define:

$$f_{x/m}(n) = f_x(n) \text{ ndiv } m$$

General multiplication

This is harder because the error in approximation for one number is multiplied by the magnitude of the second number. We need a first evaluation to determine their approximate magnitude.

```
let real_mul x y n =
  let n2 = n + 2 in
  let r = n2 / 2 in
  let s = n2 - r in
  let xr = x(r)
  and ys = y(s) in
  let p = log2 xr
  and q = log2 ys in
  if p = 0 & q = 0 then Int 0 else
  let k = q + r + 1
  and l = p + s + 1
  and m = p + q + 4 in
  (x(k) */ y(l)) ndiv (Int 2 **/ Int m);;
```

The error analysis is long but not really difficult.

Multiplicative inverse (1)

In order to get any sort of upper bound on the inverse, let alone a good approximation, we need to get a *lower* bound for the argument.

So first define the `msd` function:

```
let msd =
  let rec msd n x =
    if abs_num(x(n)) >/ Int 1 then n
    else msd (n + 1) x in
  msd 0;;
```

This finds the first n with $|f_x(n)| > 1$.

Note that if $x = 0$ then this will loop indefinitely. This problem is inevitable since equality of reals, like equality of functions, is not computable.

However if $x \neq 0$ the above must terminate eventually.

Multiplicative inverse (2)

Now we can code the inverse.

```
let real_inv x n =
  let x0 = x(0) in
  let k =
    if x0 >/ Int 1 then
      let r = log2 x0 - 1 in
      let k0 = n + 1 - 2 * r in
      if k0 < 0 then 0 else k0
    else
      let p = msd x in
      n + 2 * p + 1 in
  (Int 2 **/ Int (n + k)) ndiv (x(k));;
```

Now, of course, it is straightforward to define division:

```
let real_div x y =
  real_mul x (real_inv y);;
```

Ordering and equality

To decide the ordering relation of x and y it suffices to find an n such that $|f_x(n) - f_y(n)| \geq 2$. For example, if $f_x(n) \geq f_y(n) + 2$ we have

$$2^n x > f_x(n) - 1 \geq f_y(n) + 1 > 2^n y$$

and so $x > y$.

```
let rec separate n x y =
  let d = x(n) -/ y(n) in
  if abs_num(d) >/ Int 1 then d
  else separate (n + 1) x y;;
let real_gt x y = separate x y >/ Int 0;;
let real_ge x y = real_gt x y;;
let real_lt x y = separate x y </ Int 0;;
let real_le x y = real_lt x y;;
```

Note that the only way to arrive at the reflexive orderings is to justify the irreflexive version. In general, all these will fail to terminate if $x = y$.

Testing

We first define a function to show us a real:

```
let view x d =
  let n = 4 * d in
  let out = x(n) // (Int 2 **/ Int n) in
  approx_num_fix d out;;
```

Now we can test out some simple examples:

```
#let x = real_of_int 3;;
x : int -> num = <fun>
#let xi = real_inv x;;
xi : int -> num = <fun>
#let wun = real_mul x xi;;
wun : int -> num = <fun>
#view x 20;;
it : string = "3.00000000000000000000"
#view xi 20;;
it : string = ".33333333333333333333"
#view wun 20;;
it : string = "1.00000000000000000000"
```

The problem of reevaluation

The following is very slow:

```
#let x1 = real_of_int 1 in
 let x2 = real_mul x1 x1 in
 let x3 = real_mul x2 x2 in
 let x4 = real_mul x3 x3 in
 let x5 = real_mul x4 x4 in
 let x6 = real_mul x5 x5 in
 let x7 = real_mul x6 x6 in
 view x7 10;;
 - : string = "+1.0000000000"
```

The problem is the repeated evaluation of the same expression. This is worse because multiplication and inversion often evaluate their arguments more than once. This tends to lead to a blowup exponential in the depth of the expression.

Caching

We can solve the problem by giving each function a 'memo'.

It will remember the most accurate result it has already calculated. If asked for the same one, or a less accurate one, it can take it from the store.

If we know that $|f_x(n+k) - 2^{n+k}x| < 1$ then we have:

$$
\begin{aligned}
& |f_x(n+k) \text{ ndiv } 2^k - 2^n x| \\
\leq \quad & \frac{1}{2} + |\frac{f_x(n+k)}{2^k} - 2^n x| \\
= \quad & \frac{1}{2} + \frac{1}{2^k}|f_x(n+k) - 2^{n+k}x| \\
< \quad & \frac{1}{2} + \frac{1}{2^k} \\
\leq \quad & 1
\end{aligned}
$$

Hence we are always safe in returning $f_x(n+k) \text{ ndiv } 2^k$.

Memo function

We can use the following generic function to attach a memo to a function:

```
let memo f =
  let mem = ref (-1,Int 0) in
  fun n ->
    let (m,res) = !mem in
    if n <= m then
      if m = n then res
      else res ndiv (Int 2 **/ Int(m - n))
    else
      let res = f n in
      mem := (n,res); res;;
```

Now we systematically insert this in each operator. The resulting system is much more efficient.

```
let real_add f g = memo (fun n ->
   (f(n + 2) +/ g(n + 2)) ndiv (Int 4));;
...
```