Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 7

Infinite data structures

Topics covered:

- Functions as infinite data structures
- Real numbers as approximating functions
- Streams
- Examples: primes, Hamming's problem

Functions as infinite data structures

Ordinary ML data structures are always finite.

We can, however, regard a function out of an infinite type as being, at least potentially, infinite.

In a sense it is still finite, because the function is still determined by its definition, i.e. a finite rule. So its ‘information content’ is still finite.

But for most purposes we can think of functions as representing a potentially infinite data structure.

This is quite convenient in many situations.

We’ll examine two in more depth.

Real numbers

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.

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 -> int`, but really we should use a type of infinite-precision numbers.

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. We'll use our earlier function `exp x n` to raise `x` to the power `n`.

```
- fun real_of_int k n =
      (exp 2 n) * k;
> val real_of_int = fn : int -> int -> int
- real_of_int 23;
> val it = fn : int -> int
```

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:

```
  fun real_neg f n = ~(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` on integers.

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:

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

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 `div`, 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 $\frac{1}{2}$.

```
fun pdiv x y =
  if 2 * abs(x mod y) <= abs(y)
  then x mod y else x mod y + 1;

fun ndiv x y =
  let val q = pdiv (abs x) (abs y)
   in if x * y < 0 then ~q else q
  end;
```

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:

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

Accordingly we make our definition:

```
fun real_add f g n =
  ndiv (f(n + 2) + g(n + 2)) 4;
```

Streams

We can use functions to represent infinite data structures in a somewhat different way, which has more in common with finite lists.

Consider the following datatype for *streams*, i.e. finite or infinite lists.

```
datatype ('a)stream =
    Nil
  | Cons of 'a * (unit -> ('a) stream);
```

Crudely speaking, a list is represented recursively as: the n^{th} element together with a function that, when called, returns the rest of the list (from element $n + 1$ onwards).

Although the list is potentially infinite, we only ever evaluate it as far as we need. This explains the alternative name for streams of *lazy lists*. They are also often called *sequences*.

Generating infinite streams

How can we generate an infinite stream? Simply with a recursive function. For example, the list of numbers

$$n, n+1, n+2, \ldots$$

can be created by

```
-  fun from n =
    Cons(n, fn () => from(n + 1));
> val from = fn : int -> int stream
```

Note that because of the evaluation rules, a call `from k` won't loop indefinitely, because nothing is evaluated under the `fn () => ...`. We mustn't use:

```
fun cons h t = Cons(h, fn () => t);

fun from n = cons n (from(n + 1));
```

Stream operations

We can define some list operations in essentially the same way as for finite lists, e.g.

```
fun el 0 (Cons(h,t)) = h
  | el n (Cons(h,t)) = el (n-1) (t());

fun first 0 l = []
  | first n (Cons(h,t)) =
      h::(first (n-1) (t()));
```

Here is a definition analogous to `map`:

```
- fun maps f Nil = Nil
    | maps f (Cons(h,t)) =
        Cons(f h,fn () => maps f (t()));
> val maps = fn : ('a -> 'b) ->
                  'a stream -> 'b stream
- el 3 (maps (fn n => 2 * n) (from 1));
> val it = 8 : int
```

Filtering

We can also define a filtering operation on streams:

```
fun filters P Nil = Nil
  | filters P (Cons(h,t)) =
      if P h then
        Cons(h,fn () => filters P (t()))
      else filters P (t());
```

This will normally terminate, but not of course if there isn't anything satisfying the predicate:

```
- filters (fn x => x mod 2 = 1) (from 1);
> val it = Cons(1, fn) : int stream
- first 5 it;
> val it = [1, 3, 5, 7, 9] : int list
- filters (fn x => x mod 2 = 1)
    (maps (fn n => 2 * n) (from 1));
```

Example: primes

We can now generate the stream of all prime numbers:

```
fun genprimes (Cons(h,t)) =
    Cons(h,fn () =>
                genprimes
                 (filters
                    (fn x => x mod h <> 0)
                    (t())));
```

This takes the head of the list, filters all multiples of the head out of the tail, and then calls recursively on the filtered tail.

```
val primes = genprimes (from 2);
```

This does indeed give the sequence of all primes, e.g.

```
- first 15 primes;
> val it = [2, 3, 5, 7, 11, 13, 17, 19,
            23, 29, 31, 37, 41, 43, 47]
           : int list
```

The Hamming Problem (1)

The following function merges two ordered streams into another ordered stream:

```
fun merge Nil l2 = l2
  | merge l1 Nil = l1
  | merge (l1 as Cons(h1,t1))
          (l2 as Cons(h2,t2)) =
      if h1 < h2
      then Cons(h1,fn () =>
                     merge (t1()) l2)
      else Cons(h2,fn () =>
                     merge l1 (t2()));
```

We can use this to solve a problem attributed to Hamming: generate all numbers of the form $2^a 3^b 5^c$ in numerical order.

```
val p2 = maps (fn n => exp 2 n) (from 0);
val p3 = maps (fn n => exp 3 n) (from 0);
val p5 = maps (fn n => exp 5 n) (from 0);
```

The Hamming Problem (2)

We need to take all products from two streams:

```
fun prod (l1 as Cons(h1,t1))
         (l2 as Cons(h2,t2)) =
  Cons(h1 * h2, fn () =>
    let val r1 = t1()
        val r2 = t2()
     in merge
         (maps (fn x => h2 * x) r1)
         (merge (maps (fn x => h1 * x) r2)
                (prod r1 r2))
    end);
```

Now we just solve the Hamming problem by:

```
- val hamming = prod p2 (prod p3 p5);
> val hamming = Cons(1, fn) : int stream
- first 25 hamming;
> val it = [1, 2, 3, 4, 5, 6, 8, 9, 10,
     12, 15, 16, 18, 20, 24, 25, 27, 30,
     32, 36] : int list
```

Efficiency

While it's sometimes very convenient to play with infinite data structures this way, it can be very hard to visualize what goes on inside the machine, and when.

Essentially, we are doing *lazy* functional programming in ML, so our usual intuitions (even if we have any) can fail us.

A simple mistake can sometimes make a function fail to work. In any case, such functions are often inefficient, sometimes for subtle reasons.

For example, recalculation of the same result can blow up exponentially over multiple function calls. An implementation of multiplication in the reals will have this problem unless we take special measures.