6.12 *Extended example:* Modelling mathematical functions

Prior to this course, you probably associated *function* most readily with the real-valued functions from algebra and calculus. In this course, functions are a crucial element to ML programming, and they are way to reason about any sets. In this section we take up the topic of modelling real-valued functions in ML. This section requires a semester or so of calculus.

Obviously we can model mathematical functions with plain old ML functions. However, there are some operations we would like to perform on real functions—differentiation and integration, for example. For this reason we might seek other ways to model real functions.

Start with the most familiar, polynomial functions. A polynomial in x is a sum of terms with each term being x raised to a whole number power and multiplied by a real number *coefficient*. Formally, a polynomial in x of *degree* n has the form

$$\sum_{i=0}^{n} c_i \cdot x^i$$

for some real numbers $c_0, c_1, \ldots c_n$. For example

$$6x^4 + 2x^3 + 0x^2 + 12x^1 + 8x^0$$

Of course, we would normally write that as

$$6x^4 + 2x^3 + 12x + 8$$

What information do we need to store in order to model a polynomial function? It is as simple as a list of coefficients. All other information—the degree, the exponents on the individual terms—can be inferred from the structure, as long as we have reasonable assumptions about that structure. In our case, we will assume that all coefficients (up to the highest-degree term with non-zero coefficient) is present in the list, arranged from highest degree to zero degree.

```
- [6, 2, 0, 12, 8];

val it = [6,2,0,12,8] : int list
```

The simplest operation is to evaluate this function at some given x value. Instead of evaluating it in a brute-force method involving many exponentiations, we will use a more efficient approach known as Horner's rule:

$$\sum_{i=0}^{n} c_i \cdot x^i = (c_0 + x \cdot (\ldots x \cdot (c_{n-2} + x \cdot (c_{n-1} + x \cdot c_n)) \ldots))$$

Read it this way: c_n is our starting value. The first step is to multiply by x, and then we add c_{n-1}. That is our next "value so far." Going on from highest degree coefficient to lowest, we multiply our value so far by x and add the next coefficient. We can implement this using `foldl`. The seed value is 0, and the supplied function takes the next coefficient and the accumulated answer.

```
- fun evaluatePoly(coeffs, x) = foldl(fn(c, y) => c + y * x,
=                                     0.0, coeffs);

val evaluatePoly = fn : real list * real -> real

- evaluatePoly([6.0, 2.0, 0.0, 12.0, 8.0], 4.5);

val it = 2704.625 : real
```

Computing the derivative of a polynomial is also reasonably easy.

$$\frac{d}{dx}\sum_{i=0}^{n} c_i \cdot x^i = \sum_{i=1}^{n} i \cdot c_i \cdot x^{i-1}$$

In our case,

$$\frac{d}{dx}(6x^4 + 2x^3 + 12x + 8) = 24x^3 + 6x^2 + 12$$

What change would need to be computed from the list `[6.0, 2.0, 0.0, 12.0, 8.0]`? The list standing for the derivative would need to be one element shorter, specifically the last element would be removed. `12.0` would stay the same, but the other elements would need to be multiplied by factors 2, 3, 4, from back to front.

Clearly the process we are describing needs to be computed starting at the end of the list and working our way forward. In terms of recursion, this means the rest of the list would need to be processed first, then the current element. But to process the current element, we would need to know how many elements there are in the rest of the list, in order to know the factor for the current element. To handle this, we could have the function return two things: not only the transformed list, but also the factor for the next element.

```
- fun differentiatePoly([c]) = ([], 0)
=   | differentiatePoly(c::rest) =
=       let val (diffRest, degree) = differentiatePoly(rest)
=       in (real(degree) * c ::diffRest,
=           degree + 1)
=       end;

Warning: match nonexhaustive
val differentiatePoly = fn : real list -> real list * int
```

Alternately, we could use `foldr` (see Exercise 6.5.7). Often we will find that using of functions like `map`, `foldl`, and `foldr` will make our functions shorter, though less readable (unless you are used to thinking in terms of these functions). This time it does not make our program shorter, but it does eliminate the match-nonexhaustive warning and make it so the final answer is just the derivative, not a derivative/next-degree tuple.

```
- fun differentiatePoly(coeffs) =
=     #1(foldr(fn(c, (derivCoeffs, degree)) =>
=                (if degree = 0
=                    then []
=                    else real(degree) * c::derivCoeffs,
=                 degree + 1),
=              ([], 0), coeffs));

val differentiatePoly = fn : real list -> real list

- differentiatePoly([6.0, 2.0, 0.0, 12.0, 8.0]);

val it = [24.0,6.0,0.0,12.0] : real list
```

The function `indefIntegratePoly`, to compute the coefficients of a polynomial that is an indefinite integral of a given polynomial, is left for an exercise. Once computed, we can use it to compute a definite integral:

```
- fun defIntegratePoly(coeffs, min, max) =
=     let val indefIntegral = indefIntegratePoly(coeffs)
=     in evaluatePoly(indefIntegral, max) -
=        evaluatePoly(indefIntegral, min) end;

val defIntegratePoly = fn : real list * real * real -> real
```

However, our original goal was to model mathematical functions, not just the restricted class of polynomials. Different kinds of real functions would require different data storage, and the operations of evaluation, differentiation, and integration would be defined differently. Moreover, we would like to be able to use values of these different kinds of functions uniformly. In other words, we want a type `function` that all the kinds of functions will fit into.

Consider two other kinds of functions: Exponential functions in the form $c \cdot e^x$ for some constant c, and step functions, defined for a given step point v and step level c to be

$$f(x) = \begin{cases} 0 & \text{if } x < v \\ c & \text{otherwise} \end{cases}$$

If we take our usual approach of representing information with datatypes processed by functions with pattern-matching, we might come up with the following. (`Real.Math.exp` is a library function that computes powers of e.)

```
- datatype function =
=         Poly of real list | Exp of real | Step of real * real;

- fun evaluate(Poly(coeffs), x) =
=        foldl(fn(c, y) => c + - y * x, 0.0, coeffs)
=   | evaluate(Exp(c), x) =
=        c * Real.Math.exp(x)
=   | evaluate(Step(v, c), x) =
=        if x < v then 0.0 else c;

val evaluate = fn : function * real -> real
```

We want to take a different approach on this problem for two reasons. First is *extensibility*. The old way of making datatypes does not allow us to add new varieties of functions later, at least not without rewriting the old code. The second reason is *data hiding*. Code where data and operations are available to the rest of the program only on a "need to know" basis is less error-prone and easier to modify because the code is less dependent on the details of other parts of the code.

extensibility

data hiding

In our current example, suppose that our system of real functions is to be used by another system, say one that draws graphs of functions or does some sort of data analysis. The system using ours (the client system) should be able to work with functions of any kind uniformly—without knowing what kind of function it is or how the information for that kind of function is stored. That way changes to the implementation of the functions can be made, including making new kinds of functions, without any change needing to be made to the client system.

loose coupling

The way to achieve this independence (referred to as *loose coupling* between parts of code) is to have the different parts of the code interact through stable and clearly defined operations. In our case, the available operations are evaluation, finding the derivative, and computing a definite integral. (It might seem more fundamental to make *indefinite* integration an available operation, since a definite integral always can be computed from an indefinite integral. We chose the definite integral instead of the indefinite integral because for many kinds of functions the indefinite integral is hard to find; in those cases we will compute a numerical approximation of the definite integral.)

interface

Here we define the type `function` in terms of its *interface*, that is, the types of its available operations:

```
- datatype function = Func of
=  (real -> real) * (unit -> function) * (real * real -> real);
```

How then do we make values of this type? We write (ML) functions that will take the necessary information for a kind of (real) function (say, a list of coefficients for a polynomial), make appropriate (anonymous ML) functions, and return a value of type function.

```
- fun makePolynomial(coeffs) =
=     Func ((* Evaluate *)
=           fn (x) => foldl(fn(c,y) => c + y * x, 0.0, coeffs),
=           (* Find derivative *)
=           fn () =>  makePolynomial(#1(foldr(
=                      fn(c, (derivCoeffs, degree)) =>
=                         (if degree = 0
=                             then []
=                             else real(degree)*c::derivCoeffs,
=                          degree + 1),
=                      ([], 0), coeffs))),
=           (* Compute definite integral *)
=           fn (min, max) =>
=              let val Func(evaluateIntegral, _, _)  =
=                   makePolynomial(indefIntegratePoly(coeffs));
=              in evaluateIntegral(max) - evaluateIntegral(min)
=              end);

val makePolynomial = fn : real list -> function
```

This code is plenty complicated, so digest it carefully. The evaluate function is straightforward. The function for making a derivative contains our earlier code for `differentiatePoly` to make the coefficient list for the derivative. However, we no longer can return a real list—we need a value of our new function type. Accordingly, we feed that real list into a recursive call to `makePolynomial`, which will make the appropriate three operations for that new polynomial.

The function for the definite integral is similar but more complicated. As with the differentiation function (and our earlier function for integrating polynomials), we first make the indefinite integral. (We call `indefIntegratePoly` rather than put the code there directly so as not to spoil Project 6.C.) In the function returned from the recursive call to `makePolynomial`, the only component function we care about is the evaluate function, which we name `evaluateIntegral`. Finally, we compute the definite integral by evaluating the indefinite integral at the endpoints of the range and subtracting.

(These "recursive calls" to `makePolynomial` do not seem to have a base case. Why do we not have infinite recursion? Because those calls are not made by `makePolynomial` itself, but by the functions that `makePolynomial` returns, which in turn will not be called until after `makePolynomial` returns. Although `makePolynomial` is self-referential, it is not recursive in the conventional sense. It does not call itself. It merely makes functions that call it.)

Consider how to make a step function. Its derivative is the constant function 0, which is a special case of polynomial (we ignore the fact that the function is not differentiable at the the step point). Integration is just a matter of computing the area of a rectangle, but it depends on whether the step point falls after the max, before the min, or between the min and max. (If the min is greater than the max, switch them and return the opposite.)

```
- fun makeStep(stepPoint, stepLevel) =
=     Func (fn (x) => if x < stepPoint then 0.0 else stepLevel,
=           fn () => makePolynomial([0]),
=           fn (min, max) =>
=              let val (rmin, rmax, sign) =
=                      if min <= max then (min, max, 1.0)
=                                    else (max, min, ~1.0)
=              in sign * (if max < stepPoint then 0.0
=                         else if min > stepPoint
=                              then stepLevel * (max - min)
=                         else stepLevel * (max - stepPoint))
=              end);
```

Finally, we write a function to make exponential functions. You may have guessed that this kind of function was chosen as an example because of how easy the derivative and integral are.

```
- fun makeExponential(coefficient) =
=     Func (fn (x) => coefficient * Real.Math.exp(x),
=           fn () => makeExponential(coefficient),
=           fn (min, max) =>
=              coefficient * (Real.Math.exp(max) -
=                             Real.Math.exp(min)));
```

Students with experience in object-oriented programming such as in Java will recognize what we are doing in this chapter. The datatype function is equivalent to a Java interface, and the kinds of functions would be implemented in Java by classes that implement that interface. The functions `makePolynomial`, `makeStep`, and `makeExponential` stand in for classes (or constructors for those classes, depending on how you look at it). In particular, in a language like Java the body of the derivative function of `makeExponential` would be replaced by `this`.

Project

6.A Rewrite `evaluatePoly` to use "brute force" (not Horner's rule) in computing the value of a polynomial function for a given x. That is, given $3x^2 - 2x + 3$, start with 0 as a running sum. Then add 3. Then compute $-2x$ and add, then compute $3x^2$ and add. Use `foldr`. It is not necessary to compute the powers of x directly (such as using `Math.pow`). Instead, the anonymous function passed to `foldr` should return two things: the "answer so far" and the next or current power of x. That way each step requires only two multiplications (multiplying x by the previous power of x and multiplying the coefficient by the current power of x) and an addition, rather than an exponentiation and an addition.

6.B Write the function `indefIntegratePoly` which takes a list of coefficients standing for a polynomial and returns a new list of coefficients standing for an indefinite integral of that polynomial. Use `foldr`.

6.C Write a function `secantMethod` that takes a function (our second definition of that type) and returns an ML function to evaluate the given function's derivative at a point. The secant method approximates a derivative at a point by choosing a nearby point on the (real) function and computing the slope between those two points. Notice that the (ML) function you are asked to write does not return a function but just the evaluate portion of the derivative (type real $\rightarrow$real).

6.D Write a function `trapezoidMethod` that takes a function and a min and max value and computes an approximation of the definite integral of that function using the trapezoid method. The trapezoid method divides the area under the curve of the function into segments and approximates their areas with trapezoids.

6.E Write a function `makeSum` that takes two functions, say f and g, and returns a new function to stand for the function $h(x) = f(x) + g(x)$. Recall that $\frac{d}{dx}[f(x) + g(x)] = f'(x) + g'(x)$ and $\int f(x) + g(x)\ dx = \int f(x)\ dx + \int g(x)\ dx$.

6.F Write a function `makeProduct` that takes two functions, say f and g, and returns a new function to stand for the function $h(x) = f(x) \cdot g(x)$. Use the product rule (and `makeSum` and `makeProduct` for differentiation and your `trapezoidMethod` for integration).

6.G Write a function `makeQuotient` that takes two functions, say f and g, and returns a new function to stand for the function $h(x) = f(x)/g(x)$. Use the quotient rule (and various function-making functions) for differentiation and your `trapezoidMethod` for integration.

6.H Write a function `makeArbitrary` that takes an ML function (type real $\rightarrow$real), assumed to model a real-valued mathematical function, and returns a function to model that same mathematical function. Use `secantMethod` for differentiation and `trapezoidMethod` for integration.

6.13 *Special Topic:* Countability

Both our informal definition of cardinality in Section 1.13 and the more careful one in Chapter 6.6 were restricted to finite sets. This was in deference to an unspoken assumption that the cardinality of a set ought to be *something*, that is, a whole number. As has been mentioned already, we cannot merely say that a set like $\mathbb{Z}$ has cardinality infinity. Infinity is not a whole number—or even a number at all, if one has in mind the number sets $\mathbb{N}$, $\mathbb{W}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$. The definition of cardinality taken at face value, however, does not guarantee that the cardinality of a set is something; it merely inspired us to define what the operator $||$ means by comparing a set to a subset of $\mathbb{N}$. Indeed, the definition of cardinality merely gives us a way