Today, we’ll be talking about two important things functional programming allows you to do:

• Use the type system to communicate. We’ll talk about options and domain-specific datatypes as ways of doing this.

• Abstract repeated patterns of code. We’ll see our first examples of higher-order functions, which are an extremely powerful way of doing this. Clean and elegant code communicates ideas faster and better.

1 Types

Make illegal states unrepresentable. — Yaron Minsky, Jane Street Capital

1.1 Option Types

We’re going to start out by talking about option types. These were also covered in lab last week, but we’re going to go over some more examples today. Recall that options are a simple datatype defined for us in SML by something that looks like this:

datatype 'a option =
    NONE
  | SOME of 'a

Here again is a type parameterized over the type variable 'a, so this means there is a type t option for every type t. The same constructors can be used to construct values of different types. For example

val x : int option = SOME 4
val y : string option = SOME "a"

This is because NONE and SOME are polymorphic: they work for any type 'a.

NONE : 'a option
SOME : 'a -> 'a option

*based on notes by Brandon Bohrer and others
1.1.1 Boolean Blindness

If you ask someone “Do you know what time it is?”, you usually expect them to answer one of two ways: either “Yes, it’s [time]” or “No, sorry!”1. Despite asking what appears to be a yes or no question, you would be a bit taken aback if someone answered simply “Yes”, but this is essentially what you’re expecting if you phrase as a t -> bool function something that should really be t1 -> t2 option.

(* REQUIRES: true *)
do_you_know_what_time_it_is? : person -> bool
(* REQUIRES: person knows the time *)
tell_me_the_time : person -> int

case (do_you_know_what_time_it_is? p) of
  true => tell_me_the_time p
| false => ...

The problem is that there’s nothing about the code that prevents you from also saying tell_me_the_time in the false branch. The specification of tell_me_the_time might not allow it to be called—but specs are not automatically checked.

On the other hand, if you say

(* REQUIRES: true *)
what_time_is_it : person -> int option

case (what_time_is_it p) of
  SOME t => ...
| NONE => ...

you naturally have the time in the SOME branch, and not in the NONE branch, without making any impolite demands. The structure of the code helps you keep track of what information is available—this is easier than reasoning about the meaning of boolean tests yourself in the specs. And the type system helps you get the code right.

1.1.2 T options are not Ts

In ML, the type system reminds you that what_time_is_it might fail, and forces you to case on a value of option type. You can’t treat a T option as a T, and write something like

"The time is " ^ what_time_is_it Chris

when what_time_is_it Chris is actually NONE. You have to explicitly pass from the T option to the T by case-analyzing, and handle the failure case.

Languages like C and Java have the dreaded null pointer: when you say String in Java, or int* in C, what that means is “maybe there is a string at the other end of this pointer, or maybe

---

1I’ve lived in Pittsburgh for six years now, and have become accustomed to this sort of politeness. I’m originally from New York, where you might have to expect silence or another answer which it would be inappropriate to write in these lecture notes.
not”. You always have to remember to check the result without any mechanical reminder to do so, which is exactly the sort of thing that humans are excellent at forgetting.

Sir Charles Antony Richard Hoare\(^2\)—the inventor of QUICKSORT, ALGOL, Hoare Logic, and countless other brilliant ideas—calls this his “billion dollar mistake”. Not rigidly enforcing the distinction between things and potential things has

“led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years”.

The really beautiful thing about ML and other languages that have options it that you can use the type system to track these obligations—and at compile time, no less. If you say something has type \( T \), you know that you definitely absolutely have a \( T \). If you say \( T \) option, you know that maybe you have one, or maybe not—and the type system forces you to handle either possibility.

Thus, the distinction between \( T \) and \( T \) option lets you make appropriate promises, using types to communicate.

1.2 Domain-Specific Datatypes: Polynomials

If you type a query like \((x + 2)^2 = x(x + 4) + 4\) into Wolfram Alpha, it will say “yes, \((x + 2)^2 = x(x + 4) + 4\)”. How does it do that?

1.2.1 A Datatype for Polynomials

To illustrate programming with domain-specific datatypes, we’re going to represent polynomials and define an equality test for them. Remember from high school that a (univariate) polynomial is something like \( x^2 + 2x + 1 \), built up using the variable \( x \), constants, addition, and multiplication.

How should we represent polynomials?

One option is strings. The problem with this is that we’d then be dealing with the details of strings like “\(x^2+2x+1\)” all over the code.

Another idea is to use coefficient lists, which we’ll talk about below. The problem with coefficient lists is that they are too lossy: we want to maintain enough information about what the user typed in that we can respond, “yes, \((x + 2)^2 = x(x + 4) + 4\)”. Both of these have the same coefficient list, so we would say ‘yes, \(x^4 + 4x + 4 = x^2 + 4x + 4\)”.

A middle ground is to use a datatype to capture the important features of the problem domain, abstracting away from the concrete textual representation, but preserving some of the structure of the expression that was typed in. We can represent polynomials with a datatype as follows:

```
data type poly =
    X
  | Const of int
  | Plus of poly * poly
  | Times of poly * poly
```

Unlike lists or trees, this is a domain-specific datatype: you’d only use it if you were programming with polynomials.

Values are constructed by applying the datatype constructors. For example, \(x^2 + 2x + 1\) is represented by

\(^2\)“Tony,” to his friends.
\[ \text{Plus}(\text{Times}(X, X), \text{Plus}(\text{Times}(\text{Const } 2, X), \text{Const } 1)) \]

This represents the *abstract syntax* of an expression as a tree, whose nodes label an operation (plus, times), and whose subtrees are the arguments to the operation.

Just like lists and trees, we can define functions using pattern matching and recursion. Here’s how you apply a polynomial to an argument:

\[
\text{fun apply} \ (p \ : \ \text{poly}, \ a \ : \ \text{int}) \ : \ \text{int} = \\
\quad \text{case} \ p \ \text{of} \\
\quad \quad X \ => \ a \\
\quad \quad \text{| Const } c \ => \ c \\
\quad \quad \text{| Plus} (p1, \ p2) \ => \ \text{apply} \ (p1, \ a) + \ \text{apply} \ (p2, \ a) \\
\quad \quad \text{| Times} (p1, \ p2) \ => \ \text{apply} \ (p1, \ a) \ * \ \text{apply} \ (p2, \ a)
\]

For example, \( \text{apply}(\text{Times}(\text{Plus}(X, \text{Const } 1), \text{Plus}(X, \text{Const } 1)), 4) \) computes to 25, as you would expect. There are four cases, corresponding to the four *datatype constructors*. There are two recursive calls in the Plus and Times cases, corresponding to the two recursive references in the datatype definition.

### 1.2.2 Equality

How would you test whether two polynomials are equal? First, what does “equality” even mean? For example, you know that \((x + 1)^2 = x^2 + 2x + 1\). These two polynomials are not syntactically the same—one starts with a Plus and the other a Mult. What we mean by equality is that they *agree on all arguments*.

We can’t test this using just apply: we’d have to apply them to infinitely many arguments.

However, you learned how to do this in high school: put the polynomials in *normal form*

\[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \ldots \]

and then compare the coefficients.

What we’re going to do is to write a program to normalize a polynomial.

It is convenient to represent normal forms using a different type than poly, as lists of coefficients. E.g. \([c_0, c_1, c_2, c_3, \ldots] \) means

\[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \ldots \]

(* represent \( c_0 \ x^0 + c_1 \ x + c_2 \ x^2 + \ldots \)

by the list \([c_0, c_1, c_2, \ldots] \)

empty list is 0

*)

type \( \text{nf} \) = int list

Now, we write

\[
\text{val norm : poly} \to \ \text{nf}
\]

Working backward, we’ll write norm as follows: we interpret each syntactic constructor as an operation on normal forms. The operations we need are
val xnf : nf
val constnf : int -> nf
val plusnf : (nf * nf) -> nf
val timesnf : (nf * nf) -> nf

Some of these are very short:

val xnf : nf = [0,1]
fun constnf (c : int) : nf = [c]

For addition, we just add the coefficients, or return the longer polynomial if one is zero:

fun plusnf (n1 : nf , n2 : nf) : nf =
  case (n1, n2) of
    ([] , n2) => n2
  | (n1 , []) => n1
  | (c1 :: cs1 , c2 :: cs2) => (c1 + c2) :: plusnf (cs1 , cs2)

Multiplication is trickier. You can write it out explicitly, though we didn't do so in lecture. However, there is a nicer way to do it once we have higher-order functions.

The idea is FOIL:

\[(x + y)(z + w) = xz + xw + yz + yw\]

That is, we take the first summand in the first polynomial times the second polynomial, then the second summand in the first times the second. In general, it's the first summand in the first times the second, then the rest of the first times the second.

Here's how we implement it:

fun timesnf (n1 : nf , n2 : nf) = 
  let
    (* Purpose: multiply each number in the list by c' *)
    fun multAll (n : nf , c' : int) : nf =
      case n of
        [] => []
      | c :: cs => (c * c') :: multAll (cs , c')

    (* Purpose: compute \((c x^e) * n\) *)
    fun mult1 (n : nf, c : int , e : int) : nf =
      case e of
        0 => multAll (n , c)
        (* correct because \(c x^0 * n\) is \(c * n\) *)
      | _ => 0 :: mult1 (n , c , e - 1)
        (* correct because
          \(1) c x^e * n = x*(c x^{(e-1)} * n)
          \(2) for the coefficient list representation
              x*n can be implemented by 0 :: n
          *)
  in
    mult1 (n1 , c2 , e)
  end


(* if n1 = [c0,c1,c2,...] 
   compute (c0x^e + c1 x^(e + 1) + ...) * n2
   *)
fun times' (n1 : nf , n2 : nf, e : int) =  
case n1 of 
  [] => []
| c1 :: cs1 => plusnf (mult1 (n2 , c1 , e),
   times' (cs1 , n2 , e + 1))
   (* i.e. (c1 x^e)*n2 + (c1 x^(e+1)*n2 + c2 x^(e+2)*n2 + ...) *)
in  times' (n1 , n2 , 0)
end

This function is long, but not difficult, if we break it down: multAll just multiplies each thing in a list by a coefficient. mult1 multiplies a normal form by a single term cx^e. Note that, for the coefficient representation, x * n is implemented by just consing on 0 and shifting everything else down. times' just keeps track of the current exponent, and does what we said above: multiple the second polynomial by the first term, and then recursively by the rest. Finally, times starts the exponent off at zero.

We can put this all together into the structural normal-form function we imagined as

Caveat: the normal forms we have computed so far are not quite unique: [1,2,1,0] and [1,2,1] both represent 1 + 2x + x^2, though the former with a gratuitous 0x^3. So to finish things off, we’d need to remove trailing zeroes, which we’ll leave as an exercise.

2 Functions as Arguments

Next, we’re going to talk about the thing that makes functional programming *functional*:

Functions are values that can be passed to, and returned from, other functions.

This is an extremely powerful idea and allows us to write a new class of functions. Most of the functions we’ve seen so far are *first-order*: they accept and return non-function data types. Today, we’ll start seeing *higher-order* functions. Today we’ll look specifically at functions that accept other functions as arguments. Tomorrow we’ll start looking at higher-order functions that return functions as results.

2.1 map

Consider the following two functions:

fun double (x : int) : int = x * 2
fun doubAll (l : int list) : int list =  
case l of 
  [] => []
| x :: xs => double x :: doubAll xs
fun raiseBy (l : int list , c : int) : int list =
    case l of
        [] => []
    | x :: xs => (x + c) :: raiseBy (xs, c)

How do they differ? They both perform the same fixed transformation uniformly to each element
of a list, but they do different things—adding \( c \), multiplying by \( 2 \). Let’s write one function that
expresses the pattern that is common to both of them:

fun map (f : int -> int , l : int list) : int list =
    case l of
        [] => []
    | x :: xs => f x :: map (f , xs)

The idea with \( \text{map} \) is that it takes a function \( f : \text{int} \rightarrow \text{int} \) as an argument, which represents
what you are supposed to do to each element of the list.

\( \text{int} \rightarrow \text{int} \) is a type and can be used just like any other type in ML. In particular, a function
like \( \text{map} \) can take an argument of type \( \text{int} \rightarrow \text{int} \) and, as we’ll see next time, a function can return
a function as a result.

While we’re at it, there’s nothing special about the type \( \text{int} \): as we saw in Friday’s lecture,
nothing in the code deals specifically with ints, so we can make the function polymorphic:

fun map (f : 'a -> 'b , l : 'a list) : 'b list =
    case l of
        [] => []
    | x :: xs => f x :: map (f , xs)

For example, we can recover \( \text{doubAll} \) like this:

fun doubAll (l : int list) : int list = map (double , l)

If you substitute \( \text{double} \) into the body of \( \text{map} \), you see that it results in basically the same code as
before.

2.2 Anonymous functions

Another way to instantiate \( \text{map} \) is with an anonymous function, which is written \( \text{fn} \ x \Rightarrow 2 * x \):  

fun doubAll (1 : int list) : int list = map (fn x => 2 * x , 1)

The idea is that \( \text{fn} \ x \Rightarrow 2 * x \) has type \( \text{int} \rightarrow \text{int} \) because assuming \( x : \text{int} \), the body \( 2 * x : \text{int} \).
To evaluate an anonymous function applied to an argument, you plug the value of the argument in
for the variable like always:

\[
\begin{align*}
(fn \ x \Rightarrow 2 * x) \ 3 \\
|\rightarrow 2 * 3 \\
|\rightarrow 6
\end{align*}
\]
It's important to note that the value of \( \text{fn } x \Rightarrow x + (1 + 1) \) is \( \text{fn } x \Rightarrow x + (1 + 1) \). You don't evaluate the body until you apply the function, because how could you? You must first know what value the variable \( x \) stands for before you can do something with the function. Values of function types are tools for transforming data in the exact same sense that values of the natural number type are tools for counting things.

\( \text{doubAll} \) can be defined anonymously too:

\[
\text{val doubAll : int list} \Rightarrow \text{int list} = \text{fn } l \Rightarrow \text{map (fn } x \Rightarrow 2 \ast x , l) \]

### 2.3 Closures

A somewhat tricky, but very very useful, fact is that anonymous functions can refer to variables bound in the enclosing scope. This gets used when we instantiate \( \text{map} \) to get \( \text{raiseBy} \):

\[
\text{fun raiseBy (l , c) = map (fn } x \Rightarrow x + c , l) \]

The function \( \text{fn } x \Rightarrow x + c \) adds \( c \) to its argument, where \( c \) bound as the argument to \( \text{raiseBy} \). For example, in

\[
\text{fun raiseBy (\[1,2,3\] , 2) = map (fn } x \Rightarrow x + 2 , \[1,2,3\])}
\]

the \( c \) gets specialized to \( 2 \). If you keep stepping, the function \( \text{fn } x \Rightarrow x + 2 \) gets applied to each element of \( [1,2,3] \). The important fact, which takes some getting used to, is that the function \( \text{fn } x \Rightarrow x + 2 \) is \textit{dynamically generated}: at run-time, we make up new functions, which do not appear anywhere in the program's source code!

Here's a puzzle: what is the value of \( z \) in the following expression?

\[
\text{val z =}
\text{let}
\text{val x = 3}
\text{val f = fn } y \Rightarrow y + x
\text{val x = 5}
\text{in}
\text{f 10}
\text{end}
\]

Well, you know how to evaluate \textit{let}: you evaluate the declarations in order, substituting as you go. So, you get
let val x = 3
   val f = fn y => y + 3
   val x = 5
in
   f 10
end

The fact that \( x \) is shadowed below is irrelevant; the result is 13. This is one of the reasons why we've been teaching you the substitution model of evaluation all semester; it explains tricky puzzles like this in a natural way.

2.4 foldr

Let's look at another common pattern of recursive function we've written:

fun sum (l : int list) : int = 
case l of
  [] => 0
| x :: xs => x + (sum xs)

fun join (l : string list) : string = 
case l of
  [] => ""
| x :: xs => x ^ (join xs)

fun concat (l : 'a list list) : 'a list = 
case l of
  [] => []
| x :: xs => x @ (concat xs)

In each case, we're basically pulling apart the structure of the list and replacing all the conses with some other operator and the empty list with the unit of that operator.

sum 1 :: 2 :: 3 :: 4 :: 5 :: []
==> 1 + 2 + 3 + 4 + 5 + 0

join "a" :: "b" :: "c" :: "d" :: []
==> "a" ^ "b" ^ "c" ^ "d" ^ ""


There's a higher-order function for this too, called foldr (the r stands for “right”\(^3\)).

fun foldr (f : 'a * b -> 'b, b : 'b, l : 'a list) : 'b =

\(^3\)You might think there would also be a foldl. You would be correct.
\[
\text{case } l \text{ of}
\]
\[
\begin{align*}
& [] \Rightarrow b \\
\mid x :: xs \Rightarrow f(x, \text{foldr}(f, b, xs))
\end{align*}
\]

(Note that, in all of our examples above, we had \('a = 'b\), but the type of \text{foldr} is even more general than that!)

\[
\begin{align*}
\text{fun sum } l & = \text{foldr}(\text{op+}, 0, l) \\
\text{fun join } l & = \text{foldr}(\text{op"}, "", l) \\
\text{fun concat } l & = \text{foldr}(\text{op@}, [], l)
\end{align*}
\]

Here, we’ve used a convenient shorthand that SML gives us: the syntax \text{op+} allows us to use the + infix operator as a regular old function value. It’s equivalent to \text{fn} (a, b) \Rightarrow a + b.