1 Multiplying normal forms, higher order

Yesterday, we defined a function \texttt{timesnf} to multiply normal-form polynomials, represented as integer lists. This code was pretty complicated with a bunch of helper functions. The helper functions all just do pretty straightforward transformations on lists, which is why they were so easy to understand. We can use higher-order functions to make this code a lot shorter and more elegant!

\[
\text{fun timesnf \ (n1 : nf, n2 : nf) =}
\]
\[
\text{let}
\]
\[
\text{(* Purpose: multiply each number in the list by c' *)}
\]
\[
\text{fun multAll \ (n : nf, c' : int) : nf =}
\]
\[
\quad \text{map \ ((fn \ c \Rightarrow \ c \ast c'), \ n)}
\]
\[
\text{in}
\]
\[
\text{(* if n1 = [c0,c1,c2,...]}
\]
\[
\quad \text{compute \ (c0 + c1 \cdot x + ...) \ast n2}
\]
\[
\quad (*)
\]
\[
\quad \text{foldr \ ((fn \ (c1, \ rest) \Rightarrow \ plusnf \ (\text{multAll} \ (n2, \ c1), \ 0::\text{rest})),}
\]
\[
\quad \quad \text{\text{constnf} \ 0,}
\]
\[
\quad \quad \text{n1)}
\]
\[
\text{end}
\]

2 Currying

Moses Schönfinkel was a Russian logician and mathematician working on something called combinatory logic in the early twentieth century. In his work, he was formalizing a language that included the idea of functions, and found that it was easier to prove things about his language if he assumed every function took exactly one argument and there were no pairs. The observation that made this sound was that you can emulate a function that takes multiple arguments all at once with a function that takes the first argument and produces a function expecting the next argument.

*based on notes by Brandon Bohrer and others*
Haskell Curry was an American logician and mathematician working on something called combinatory logic in the early twentieth century. He made a similar observation and, unfortunately for Schönfinkel, his name is a lot easier to spell, write, and pronounce, so this observation is called Currying, not Schönfinkelization.¹

2.1 Example: map

This observation can be carried out in SML just as well. For example, recall the map we wrote on lists last time:

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

This gives \( \text{map} : ((\text{'a} \to \text{'b}) \times \text{'a} \text{ list}) \to \text{'b} \text{ list} \). We can write a curried version of map by first taking the first argument and then producing a function expecting the rest:

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

This gives \( \text{map} : ('a \to 'b) \to ('a \text{ list} \to 'b \text{ list}) \). By convention, the type constructor \( \to \) associates to the right, so we could have also just said

\[
\text{map} : ('a \to 'b) \to ('a \text{ list} \to 'b \text{ list})
\]

Instead of taking two arguments, a function and a list, this version of map takes one argument (a function), and returns a function that takes the list and computes the result.

Here’s a small evaluation trace to demonstrate how this new map works (with some parentheses added for clarity):

```
map (fn x => x + 1)
|-> fn l => case l
    of [] => []
    | x::xs => ( (fn x => x + 1) x)::(map (fn x => x + 1) xs)
```

And that’s it! Why do we stop? We’ve reached a value. Since there are no arguments available, there’s no more work to do. What we get is a function that’s waiting for a list is ready to add one to every element. We never wrote this function in our source code; rather, it was created from a partial application of map.

If instead we’d given map both of its arguments, we’d get something that looked like this:

¹I’m pretty sure that this, not profound economic and cultural upheaval after and in someways before the second World War, caused the Cold War.
(map (fn y => y * 2)) (10::[])  
|→ (fn L => case L of [] => []  
  | x::xs => (((fn y => y * 2) x) :: (map (fn y => y * 2)) xs) (10::[]))  
|→ case 10::[] of [] => []  
  | x::xs => (((fn y => y * 2) x) :: (map (fn y => y * 2)) xs  
|→ ((fn y => y * 2) 10) :: (map (fn y => y * 2)) []  
|→ 10 * 2 :: (map (fn y => y * 2)) []  
|→ 20 :: (map (fn y => y * 2)) []  
|→ 20 :: (fn L => case L of [] => []  
  | x::xs => ((fn y => y * 2) x) :: (map (fn y => y * 2)) xs) []  
|→ 20 :: (case [] of [] => [] | x::xs => ...)  
|→ 20 :: []

Because currying turns out to be so useful and omnipresent, there is special version of fun declarations for curried functions. Remember that we want fun declarations to look similar to function calls, so the way we declare a curried function is by separating the arguments with spaces:

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

This is exactly the same as the explicit fn binding in the body we saw above.

Note that function application left-associates, so f x y gets parsed as (f x) y. Thus, a curried function can be applied to multiple arguments just by listing them in sequence, like map (fn x => x + 1) [1,2]. This is really two function applications: one of map to (fn x => x + 1), producing a function of type 'a list -> 'b list; and one of map (fn x => x + 1) to [1,2].

2.2 Okay, But Why Bother?

Thus far, currying may seem under-motivated: is it just a trivial syntactic transformation? We can do this, but why do we want to? There are several advantages of curried functions; here’s the big picture:

• It is easy to partially apply a curried function, e.g. you can write map f, rather than fn l => map (f,l), when you want the function that maps f across a list. This will come up when you want to pass map f as an argument to another higher-order function, or when you want to compose it with other functions, as we will see below. This is a difference in readability, but not functionality. For example, yesterday’s function doubleAll could be written more concisely as: val doubleAll: int list -> int list = map double

• Currying you write programs that you wouldn’t otherwise be able to write. Roughly, the idea is that sometimes you can do meaningful work without knowing all of the arguments. Currying gives you a way to say “if you give me just the first argument, I’ll consume it entirely and produce a function that’s waiting for the next”. For example, maybe we want to write a function that copies one file to another:

fun copy (src : filename) : filename -> result =
let
  val contents : string = read_file src
in
  fn dst : filename => write (contents, dst)
end

This is called *staged computation*. If done carefully, this can have a major effect on the performance of your program. In the example above, reading a file from disk is very expensive, so if we wanted to make many copies of the file src, we could write `val copy_src: filename -> result = copy src` and call the `copy_src` function every time we want to make a copy of src, which will allow us to make many copies, but only open the file once. In summary, staging lets you insert real work between the arguments to the function, controlling what happens when.

- In the last week of the course, we will talk about *function-local state*. It turns out that currying is essential if we want functions to be stateful.

### 2.3 Patterns Of Recursion on Trees

Higher-order functions express patterns of recursion. For example, `map` is expresses the pattern of recursion where we apply the same operation to every element of a list. Every time we have a template for writing recursive functions, we can turn it into a higher-order function.

This is true not only for lists, but also for trees. Recall the datatype declaration for polymorphic trees:

```haskell
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

2.3.1 map

We just wrote a curried version `map` for lists. Now let’s write a curried `map` for trees:

```haskell
fun treemap (f : 'a -> 'b) (t : 'a tree) : 'b tree =
  case t of
    Empty => Empty
  | Node (l, x, r) => Node (treemap f l, f x, treemap f r)
```

This applies `f` to the data at each `Node`.

### 3 Combinators

Earlier we wrote `val doubleAll: int list -> int list = map double`. This was our first taste of something called *point-free style*. The idea behind point-free style is that you don’t need to name arguments when you can just make functions! Here we have chosen to never name the argument shrub. It does get named though—transparently through what the `fun` keyword expands to.

In this section, let’s see how far we can take the above observation about point-free style.

In functional programming, one way to think about problems is to decompose them into a series of tasks, represented as functions, and to solve the problem by composing these functions together.

Function composition is a builtin operator named `o`, which is defined as follows:
infix o
fun (g : 'b -> 'c) o (f : 'a -> 'b) : 'a -> 'c =
  fn x => g (f x)

Now, anytime we might have written fun h x = g (f (x)), we can be extra cool and just write val h = g o f

Another benefit of higher-order functions is that one may “lift” operations from some type to functions that map into that type.

For instance, if \( f \) and \( g \) are two functions mapping into the integers, say of type \( t \rightarrow \text{int} \) for some type \( t \), then we may combine \( f \) and \( g \) as if they were themselves integers.

As a first example, let’s define a higher-order function \( ++ \) that adds \( f \) and \( g \) to produce a new function \( f \++ g \) of type \( t \rightarrow \text{int} \). The higher-order function \( ++ \) is an instance of what is sometimes called a \textit{combinator}.

How does one add integer-valued functions? Using the \textit{pointwise-principle} from math: \( f \++ g \) is the function whose value at a given “point” \( x \) is \( f(x) + g(x) \). This is easy to express in SML as:

\[
\begin{align*}
infixr ++ \\
\text{fun (f ++ g) (x : 'a) : int = f(x) + g(x)}
\end{align*}
\]

If we have these declarations:

\[
\begin{align*}
\text{fun square (x:int):int = x*x} \\
\text{fun double (x:int):int = 2*x}
\end{align*}
\]

then \texttt{square} represents the math function \( x^2 \) and \texttt{double} represents the math function \( 2x \). The following declaration

\[
\text{val quadratic = square ++ double}
\]

would therefore produce an SML function called \texttt{quadratic} that represents the math function \( x^2 + 2x \).

Or, suppose we define the combinator \texttt{MIN} by

\[
\text{fun MIN (f, g) (x : 'a) : int = Int.min(f(x), g(x))}
\]

Now consider

\[
\text{val lowest = MIN(square, double)};
\]

The SML function \texttt{lowest} represents the lower envelope of the two graphs of \( x^2 \) and \( 2x \). (Graph these functions and you will see!)
4 Equivalence

So, the takeaway message from the last lecture and a half (and the rest of the course) has been that functions are values. But we haven’t talked much about how extensional equivalence should deal with the fact that functions are values².

Recall, in our discussion of extensional equivalence, we said that two expressions of type \( t \) are equivalent if and only if they evaluate to the same value of type \( t \), or they both raise the same exception, or they both fail to terminate.

This is quite well-defined for equality types, but when do two values of function type “evaluate to the same value”? Suppose we have

```sml
val f = fn x => x + x
```

and

```sml
val g = fn x => 2 * x
```

Is it the case that \( f \sim g \)? Both are values, but what does it mean to ask whether \( f \) and \( g \) are the same value? You can think of extensional equivalence as meaning “will we ever be able (by putting them in the context of a larger SML program) be able to tell the two values apart?” So \( f \) and \( g \) are not the same value in a syntactic sense, but there’s no program we can write that can tell them apart, because the only thing you can do with a function is apply it to an argument.

With that intuition in mind, we can extend the definition of extensional equivalence to say that two function values \( f \) and \( g \) of type \( t \rightarrow t' \) are extensionally equivalent if and only if, for all values \( x \) of type \( t \), \( f(x) \) and \( g(x) \) are equivalent. Symbolically, \( f \cong g \) iff for all \( x : t \), \( f(x) \cong g(x) \).

Some examples of equivalence for function expressions of type \( \text{int} \rightarrow \text{int} \):

- \((\text{inc} \circ \text{double}) \cong \text{fn x:}\text{int} \rightarrow (2*\text{x}+1)
- (\text{double} \circ \text{inc}) \cong \text{fn x:}\text{int} \rightarrow 2*(\text{x}+1)
- \text{add2} \cong (\text{fn y:}\text{int} \rightarrow \text{inc(inc y)})
- \text{add4} \cong (\text{fn z:}\text{int} \rightarrow \text{add2(add2 z)})
- (\text{fn x:}\text{int} \rightarrow \text{x}+1) \cong (\text{fn w:}\text{int} \rightarrow \text{w}+1)

5 This might be important next lecture...

Yesterday, we saw the higher-order function \text{foldr} \, which combines the elements of a list from right to left. We might also want a function that combines the elements from left to right. It turns out that, to do this efficiently, we can use the same trick we used to make \text{reverse} efficient: using an accumulator argument.

We want a function \text{foldl} : (\text{a} \times \text{b} \rightarrow \text{b}) \rightarrow \text{b} \rightarrow \text{a} \text{ list} \rightarrow \text{b}. The accumulator we want is just intermediate values of the combining function, so it will be of type \( \text{b} \). We could do the same thing as with \text{fastRev} and use a helper function with an extra argument of type \( \text{b} \), but we don’t have to, since \text{foldl} already takes an argument of that type that we can use for this purpose.

²Have I mentioned that functions are values?
fun foldl (f: 'a * 'b -> 'b) (b: 'b) (l: 'a list) : 'b =
  case l of
    [] => b
  | x :: xs => foldl f (f (x, b)) xs

TO BE CONTINUED