Church Encoding

1 Lists and fold

Recall that an `'a list` is either

- the empty list, or
- a value of type `'a` prepended to an `'a list`,

and nothing else. We can translate this into an OCaml type definition

```
type 'a list = Empty | Cons of 'a * 'a list
```

where we have special syntax for the constructors: `[]` ≡ `Empty` and `x::xs` ≡ `Code (x, xs)`.

```
let rec list_func l = match l with
  | [] -> a
  | x :: xs -> g x (list_func xs)
```

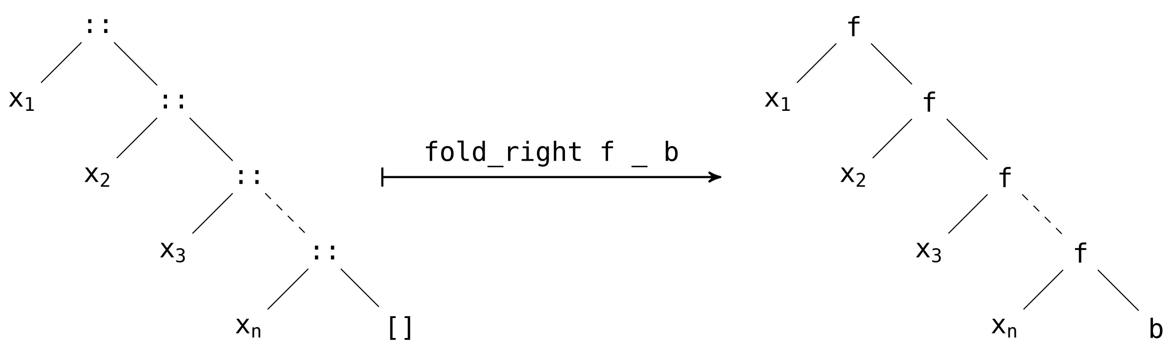
To define a function `list_func` that operates on lists, we perform recursion on the *structure* of the data (made easier by *pattern matching*):

- When we have `[]`, we're at the base case and want to return some value `a`
- When we have a nonempty list `x::xs`, we're at the recursive case and want to perform some operation `g` on the head `h` and the result of recursively applying the function to the tail, `list_func xs`

In fact, so many list functions are defined this way that we encapsulate this schema as a higher order function `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` which is defined like so:

```
let rec fold_right f l b = match l with
  | [] -> b
  | x :: xs -> f x (fold_right f xs b)
```

with `list_func l` ≡ `fold_right g l a`. Intuitively, `fold_right` replaces `[]` by `b` and `::` by `f`.



The idea behind **Church encoding** is the following: since the only thing we care about lists is what we eventually get after applying a function[1] to it, instead of defining a separate type with constructors, we can represent a list as a function that takes in some `f` and `b` and puts them where the constructors would be. In this view, a list `l` is "the same" as the result of evaluating[2] `fun f b -> fold_right f l b`.

1.1 Example Church-encoded lists

Let's look at some example encodings:

- The empty list

[1] A function defined in the `fold_right` sense.

[2] In the case of a language like OCaml, we'll pretend we can reduce inside the body of a function.

```
[] ⇝ fun f b -> fold_right f [] b
   ⇝ fun f b -> match [] with
                | [] -> b
                | x::xs -> f x (fold_right f xs b)
   ⇝ fun f b -> b
```

- The singleton list `[x]`

```
[x] ⇝ fun f b -> fold_right f [x] b
    ⇝ fun f b -> match [x] with
                 | [] -> b
                 | y::ys -> f y (fold_right f ys b)
    ⇝ fun f b -> f x (fold_right f [] b)
    ⇝ fun f b -> f x b
```

- An arbitrary list $[x_1;\ x_2;\ \ldots;\ x_n]$

$$[x_1;\ x_2;\ \ldots;\ x_n] \rightsquigarrow \texttt{fun f b -> f } x_1 \texttt{ (fold\_right f } [x_2;\ \ldots;\ x_n] \texttt{ b)}$$
$$\rightsquigarrow \texttt{fun f b -> f } x_1 \texttt{ (f } x_2 \texttt{ (f } \ldots \texttt{ (f } x_n \texttt{ b))} \ldots\texttt{)}$$

To see how we can use these encoded values, we turn our attention to the simpler type of natural numbers where we'll perform a similar procedure.

2 Numbers, naturally

First, we look at a generic function defined by recursion on natural numbers:

```
let rec nat_func n = match n with
  | 0 -> a
  | _ -> g (nat_func (n - 1))
```

Let's rewrite this in pseudo-OCaml in terms of addition instead of subtraction:

```
let rec nat_func n = match n with
  | 0 -> a
  | m+1 -> g (nat_func m)
```

It seems like the "structure" we care about when defining recursive functions on natural numbers is the fact that a number is built up by adding 1 to it some number of times — $n = 0 + \underbrace{1 + \cdots + 1}_{n \text{ times}}$. This leads to the following definition: a `nat` is either

- zero, or
- the *successor* (+1) of some existing `nat`

and nothing else. In OCaml, we translate this to the following type definition:

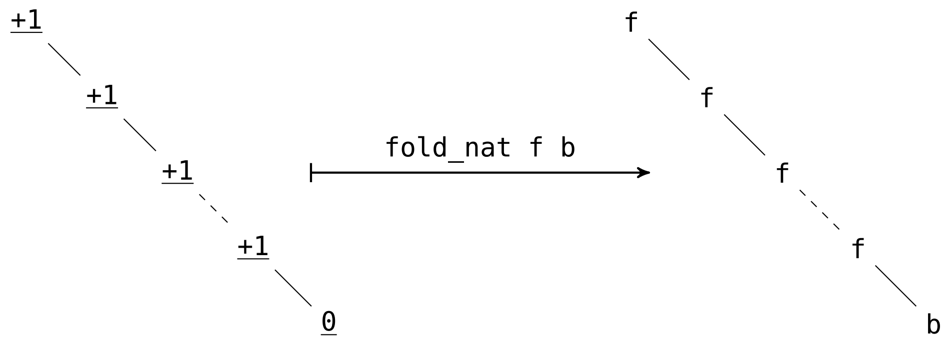
```
type nat = Zero | Succ of nat
```

In pseudo-OCaml, we'll write <u>`0`</u> ≡ `Zero` and `n`<u>`+1`</u> ≡ `Succ n`.

Abstracting over the definition schema, we get a function `fold_nat` defined by

```
let fold_nat f b n = match n with
  | 0 -> b
  | m+1 -> f (fold_nat f b m)
```

with `nat_func n` ≡ `fold_nat g a n`. The intuition is that `fold_nat` replaces <u>`0`</u> by `b` and <u>`+1`</u> by `f`:



which is to say that `fold_nat f b n` $\equiv \underbrace{\texttt{f (f ... (f}}_{\texttt{n}\ \text{times}}\texttt{ b)...)}$. For convenience, we'll write

- $\underbrace{\texttt{f (f ... (f}}_{\texttt{n}\ \text{times}}\texttt{ b)...)}$ as $\texttt{f}^{\texttt{n}}$ `b`, and
- c_n to denote the Church encoding of `n`.

Evaluating c_n `= fun f b -> fold_nat f b n` for some values of `n`, it shouldn't be too hard to see that $c_n \equiv$ `fun f b ->` $\texttt{f}^{\texttt{n}}$ `b`

2.1 Type of a Church numeral

What type does a Church-encoded number have?

- It takes in two arguments `f` and `b`

  c_n `: _ -> _ -> _`

- `f` is applied to `b`

  c_n `: ('a -> _) -> 'a -> _`

- `f` is applied to a value resulting from an application of `f`

  c_n `: ('a -> 'a) -> 'a -> _`

- The value we get at the end comes from an application of `f`

  c_n `: ('a -> 'a) -> 'a -> 'a`

Thus the type of all Church encoded natural numbers in OCaml is equivalent[3] to:

```
type 'a churchNat = ChurchNat of (('a -> 'a) -> 'a -> 'a)
```

2.2 Producing a value

2.2.1 Checking for 0

Let's write a function that checks if a given `nat` is `0`:

```
let iszero n = match n with
  | 0 -> true
  | _ -> false
```

[3]Not exactly; since the Church numeral should work uniformly over any kind of function of type `'a -> 'a`, it is actually equivalent to something like `type churchNat = forall 'a. ('a -> 'a) -> 'a -> 'a`.

Notice that this is equivalent to

```
let iszero n = match n with
  | 0 -> true
  | m+1 -> (fun _ -> false) m
```

and if we squint a little bit, we'll see that this is

```
let rec iszero n = match n with
  | 0 -> true
  | m+1 -> (fun _ -> false) (iszero m)
```

or

```
let iszero n = fold_nat (fun _ -> false) true n
```

Since `cn` is "the same" as `fun f b -> fold_nat f n b`, in order to check if a given Church numeral represents , we can translate the definition to the Church-encoded world as

```
let iszero_church (ChurchNat c) = c (fun _ -> false) true
```

Indeed,

- if `n = 0`,

```
iszero_church (ChurchNat c0)
⇝ c0 (fun _ -> false) true
⇝ (fun f b -> b) (fun _ -> false) true
⇝ (fun b -> b) true
⇝ true
```

- if `n = 1`,

```
iszero_church (ChurchNat c1)
⇝ c1 (fun _ -> false) true
⇝ (fun f b -> f b) (fun _ -> false) true
⇝ (fun b -> (fun _ -> false) b) true
⇝ (fun _ -> false) true
⇝ false
```

- if `n = 2`,

```
iszero_church (ChurchNat c2)
⇝ c2 (fun _ -> false) true
⇝ (fun f b -> f (f b)) (fun _ -> false) true
⇝ (fun b -> (fun _ -> false) ((fun _ -> false) b)) true
⇝ (fun _ -> false) ((fun _ -> false) true)
⇝ (fun _ -> false) false
⇝ false
```

- and so on.

2.2.2 Converting back to an int

We repeat the same process to figure out `church_to_int`. In the recursive version of `nat_to_int`,

- in the base case, we should get 0
- in the recursive case, we should add 1 to the result of the recursive call

or in terms of `fold_nat`,

```
nat_to_int n ≡ fold_nat (fun m -> m + 1) 0 n
```

The translation into the world of Church numerals is now straightforward.

2.3 Fun with operations

2.3.1 Addition (Take 1)

We start off by writing addition on `nat`s recursively:

```
let rec add_nat m n = match n with
  | 0 -> m
  | k+1 -> (fun x -> x+1) (add_nat m k)
```

which is equivalent to

```
let add_nat m n = fold_nat (fun x -> x+1) m n
```

Translating to the Church world, we have

```
let add (ChurchNat c_m) c = c_m add1 c
```

which means we need to figure out how to add 1 to a Church numeral.

Since c_n f is just a composition f^n, we can take advantage of properties of function composition to get what we want:

$$
\begin{aligned}
(c_{n+1}\ f)\ b &\equiv f^{n+1}\ b \\
&\equiv (f \circ f^n)\ b \\
&\equiv (f \circ c_n\ f)\ b \\
&\equiv f\ (c_n\ f\ b)
\end{aligned}
$$

which gives us

```
let add1 (ChurchNat c_n) = ChurchNat (fun f b -> f (c_n f b))
```

Note that this actually defines addition as a function of type

```
'a churchNat churchNat -> 'a churchNat -> 'a churchNat
```

which isn't exactly what we want.

2.3.2 Addition (Take 2)

Let's see what happens if we just use the properties of composition:

$$
\begin{aligned}
(c_{m+n}\ f)\ b &\equiv f^{m+n}\ b \\
&\equiv (f^m \circ f^n)\ b \\
&\equiv (c_m\ f \circ c_n\ f)\ b \\
&\equiv c_m\ f\ (c_n\ f\ b)
\end{aligned}
$$

This gives us the following definition:

```
let add (ChurchNat c_m) (ChurchNat c_n) =
  ChurchNat (fun f b -> c_m f (c_n f b))
```

that has type

```
'a churchNat -> 'a churchNat -> 'a churchNat
```

which is what we want!

2.3.3 Multiplication

Using a different property of composition:

$$
\begin{aligned}
(c_{mn}\ f)\ b &\equiv f^{mn}\ b \\
&\equiv (f^m)^n\ b \\
&\equiv c_n\ (f^m)\ b \\
&\equiv c_n\ (c_m\ f)\ b
\end{aligned}
$$

This gives us the following definition:

```
let mult (ChurchNat cm) (ChurchNat cn) =
  ChurchNat (fun f b -> cn (cm f) b)
```

3 Exercises

1. Figure out the type of a Church-encoded list.
2. Implement `isempty`, `length`, and `churchlist_to_list` for Church-encoded lists.
3. Implement `append` and `cartesian_product` for two Church lists.
4. Figure out how to exponentiate two Church numerals.
5. Pick some other datatypes and figure out their Church encodings.