Fixpoints and Recursion

Recursive definitions require self-reference. A recursive function in OCaml such as the factorial function,

```
let rec fact x = if x = 0 then 1 else x * fact (x - 1)
```

must be able to call itself recursively. We cannot do this in the λ-calculus directly, because there are no names—all functions are anonymous. In OCaml, we can fake this in the environment model using refs to create cycles:

```
let fact =
  let fact' : (int -> int) ref = ref (fun x -> x) in
  let f = fun x -> if x = 0 then 1 else x * (!fact' (x - 1)) in
  fact' := f;
  fun x -> !fact' x
```

But this still does not help, since the λ-calculus has no refs either, it is purely functional. How then can we possibly define recursive functions in the λ-calculus without names or refs? It seems hopeless. However, believe it or not, it can be done.

Approximation of Recursive Functions

To illustrate, let's use the factorial function as an example. Using our encoding of natural numbers as Church numerals developed in the last lecture, we would like to get a λ-term `fact` such that

```
fact  =  λ.(if-then-else (isZero n) 1̄ (mul n (fact (sub1 n))))
```

First, note that `fact` is a kind of limit of an inductively-defined sequence of functions $\texttt{fact}_n$, `n >= 0`, each of which can be defined without recursion.

$$\texttt{fact}_0 \;=\; \lambda n.n$$
$$\texttt{fact}_{n+1} \;=\; \lambda n.(\texttt{if-then-else}\ (\texttt{isZero}\ n)\ \bar{1}\ (\texttt{mul}\ n\ (\texttt{fact}_n\ (\texttt{sub1}\ n))))$$

Thus for any Church numeral $\bar{m}$,

$$\texttt{fact}_0\ \bar{m} \;\Rightarrow\; \bar{m}$$
$$\texttt{fact}_1\ \bar{m} \;\Rightarrow\; \texttt{if-then-else}\ (\texttt{isZero}\ \bar{m})\ \bar{1}\ (\texttt{mul}\ \bar{m}\ (\texttt{fact}_0\ (\texttt{sub1}\ \bar{m})))$$
$$\texttt{fact}_2\ \bar{m} \;\Rightarrow\; \texttt{if-then-else}\ (\texttt{isZero}\ \bar{m})\ \bar{1}\ (\texttt{mul}\ \bar{m}\ (\texttt{fact}_1\ (\texttt{sub1}\ \bar{m})))$$
$$\texttt{fact}_3\ \bar{m} \;\Rightarrow\; \texttt{if-then-else}\ (\texttt{isZero}\ \bar{m})\ \bar{1}\ (\texttt{mul}\ \bar{m}\ (\texttt{fact}_2\ (\texttt{sub1}\ \bar{m})))$$
$$\vdots$$
$$\texttt{fact}_n\ \bar{m} \;\Rightarrow\; \texttt{if-then-else}\ (\texttt{isZero}\ \bar{m})\ \bar{1}\ (\texttt{mul}\ \bar{m}\ (\texttt{fact}_{n-1}\ (\texttt{sub1}\ \bar{m})))$$

```
          .
          .
          .
```

In this definition, we have not used names in any essential way, just as an abbreviation for something that can be defined purely functionally. For example, $\texttt{fact}_2$ is equivalent to

```
λn.(if-then-else (isZero n) 1
   (mul n ((λn.(if-then-else (isZero n) 1
                (mul n ((λn.n)(sub1 n)))))(sub1 n))))
```

which reduces via the substitution model (β-reduction) to

```
λn.(if-then-else (isZero n) 1
   (mul n (if-then-else (isZero (sub1 n)) 1
          (mul (sub1 n) (sub1 (sub1 n))))))
```

By ***limit*** we mean that the functions $\texttt{fact}_n$ approximate `fact` more and more accurately as `n` gets larger, in the sense that they agree with `fact` on more and more inputs. The first approximant $\texttt{fact}_0$ agrees with `fact` on no inputs at all. The second approximant $\texttt{fact}_1$ agrees with `fact` on one input, namely `0`. The third approximant $\texttt{fact}_2$ agrees with `fact` on two inputs, namely `0` and `1`, and so on. One can show by induction that $\texttt{fact}_n$ agrees with `fact` on inputs `0`, `1`, ..., `n-1`. Although none of these approximants are equal to `fact`, they get closer and closer as `n` gets larger.

Fixpoints

Note that the inductive step in the inductive definition of $\texttt{fact}_{n+1}$ from $\texttt{fact}_n$ can be expressed abstractly as a higher-order function. If we define

```
t_fact  =  λF.λn.(if-then-else (isZero n) 1 (mul n (F (sub1 n))))
```

then our inductive definition of $\texttt{fact}_n$ can be rewritten

```
fact0   =  fun x -> x
factn+1 =  t_fact factn
```

The real factorial function `fact` (if it exists!) should satisfy the equation

```
fact  =  λn.(if-then-else (isZero n) 1 (mul n (fact (sub1 n))))
```

In other words, it should be a ***fixpoint*** of `t_fact`:

```
fact  =  t_fact fact
```

Think of `t_fact` as the operation of "unwinding" the definition of `fact` once. So if we

had a general way of obtaining fixpoints in the λ-calculus, we might apply it to obtain a fixpoint of `t_fact` and this might do the trick.

Fixpoint theorems abound in mathematics. A whorl on your head where your hair grows straight up is a fixpoint. At any instant of time, there must be at least one spot on the globe where the wind is not blowing. For any continuous map `f` from the closed real unit interval `[0,1]` to itself, there is always a point `x` such that `f(x) = x`.

The λ-calculus is no exception. It turns out that *any* λ-term `W` has a fixpoint. Consider the lambda term

```
λx.W(xx) λx.W(xx)
```

This is a fixpoint of `W`, as can be seen by performing one β-reduction step:

```
λx.W(xx) λx.W(xx)  =>  W (λx.W(xx) λx.W(xx))
```

Moreover, there is a lambda term `Y`, called the ***fixpoint combinator***, which when applied to any `W` gives a fixpoint of `W`:

```
Y  =  λw.(λx.w(xx) λx.w(xx))
```

If we apply `Y` to `t_fact`, what do we get? Define

```
fact  =  Y t_fact  =  λx.t_fact(xx) λx.t_fact(xx)
```

We know that this is a fixpoint of `t_fact`, i.e.

```
fact  =>  t_fact fact
```

Now we show by induction that this is indeed the factorial function; that is, for any `n`,

```
fact n  =>  n!
```

Basis.

```
fact 0  =>  t_fact fact 0
  =>  λn.(if-then-else (isZero n) 1 (mul n (fact (sub1 n)))) 0
  =>  if-then-else (isZero 0) 1 (mul 0 (fact (sub1 0)))
  =>  1
  =>  0!
```

Induction step:

```
fact n+1  =>  t_fact fact n+1
  =>  λn.(if-then-else (isZero n) 1 (mul n (fact (sub1 n)))) n+1
  =>  if-then-else (isZero n+1) 1 (mul n+1 (fact (sub1 n+1)))
  =>  mul n+1 (fact (sub1 n+1))
  =>  mul n+1 (fact n)
  =>  mul n+1 n!      (by the induction hypothesis)
  =>  (n+1)!
```

Note that nowhere in our development did we use names for anything but abbreviations for anonymous functions.

Encoding in OCaml

We would like to encode Church numerals and recursion without names to illustrate these constructions in OCaml. However, there are two immediate impediments to this project:

1. OCaml is typed, and the lambda calculus is not. Many of the traditional encodings given by Church do not typecheck in OCaml, even with polymorphic typing.
2. OCaml is eager, but the encoding of recursion using the traditional fixpoint combinator `Y` yields looping behavior if evaluated using an eager reduction strategy. Only lazy reduction yields normal forms.

It turns out that both these problems can be circumvented with a little extra work.

For the first, observe that some of the definitions we have given typecheck in OCaml and some do not. The fixpoint combinator `Y` definitely does not typecheck:

```
# fun w -> (fun x -> w (x x)) (fun x -> w (x x));;
Error: This expression has type 'a -> 'b
       but an expression was expected of type 'a
```

The problem here is the subexpression `(x x)`, which tries to apply `x` as a function to itself. The type inference algorithm discovers a circularity when it tries to unify the polymorphic type of `x` as a function `'a -> 'b` with the type of `x` as its own input `'a`. A similar situation arises when trying to apply a Church numeral $\overline{n}$ to a function on Church numerals such as in the definition of `add`. There is no type `s` in OCaml with the property that `s = s -> t`. However, there is something almost as good: a type `s` such that `s = Fix (s -> t)`:

```
# type 'a fix = Fix of ('a fix -> 'a);;
type 'a fix = Fix of ('a fix -> 'a)
```

Note there is no base case to the inductive definition! Nevertheless, we can construct objects of this type:

```
# Fix (fun _ -> 3110);;
- : int fix = Fix <fun>
```

Moreover, we can use such an object either as a function of type `int fix -> int` (provided we deconstruct it first using pattern matching to get rid of the `Fix`) or as an input to such a function.

Using the same idea, we can give appropriate recursive types for Church numerals:

```
type church = Ch of ((church -> church) -> church -> church)
```

For the Church numerals, the only thing we have to remember is to deconstruct it before applying it as a function. For example, instead of

```
let add1 n = fun f -> fun x -> f (n f x)
let zero = fun f -> fun x -> x
```

which is the direct translation of Church's encoding, we take

```
let add1 (Ch n) = Ch (fun f -> fun x -> f (n f x))
let zero = Ch (fun f -> fun x -> x)
```

The second problem is simulating lazy evaluation. Note that the **`if-then-else`** in the definition of `fact` must be evaluated lazily, otherwise the **`else`** clause will be evaluated prematurely. However, our lambda calculus definition of **`if-then-else`** is evaluated eagerly when we run it in OCaml. It will keep unwinding the definition of `fact`, trying to calculate better and better approximations before ever applying them, and this will go on forever. To prevent this, we use *thunks*. We wrap the **`then`** and **`else`** expressions in the body of a function to inhibit evaluation until the test has been evaluated, then evaluate the correct alternative to get the value.

Here is the encoding. Give it a try!

```
(* Church numerals *)
type church = Ch of ((church -> church) -> church -> church)

let add1 (Ch n) = Ch (fun f -> fun x -> f (n f x))
let zero = Ch (fun f -> fun x -> x)
let one = add1 zero
let two = add1 one
let three = add1 two
let four = add1 three
let five = add1 four
let six = add1 five
let seven = add1 six
let eight = add1 seven
let nine = add1 eight
let ten = add1 nine

let add n (Ch m) = m (add1) n
let mul n (Ch m) = m (add n) zero
let exp n (Ch m) = m (mul n) one

(* conditional test with thunks for lazy eval *)
(* thunks are objects of type church, although not *)
(* Church numerals *)
let thunk x : church = Ch (fun _ -> fun _ -> x)
let eval (Ch x) = x (fun z -> z) zero
let if_zero (Ch n) x y = eval (n (fun _ -> y) x)

(* pairs *)
(* pairs are also objects of type church, but not *)
(* Church numerals *)
```

```
let pair x y = Ch (fun _ -> fun z -> if_zero z (thunk x) (thunk y))
let fst (Ch p) = p (fun x -> x) zero
let snd (Ch p) = p (fun x -> x) one

(* subtraction *)
let next p = pair (snd p) (add1 (snd p))
let sub1 (Ch n) = fst (n next (pair zero zero))
let sub n (Ch m) = m sub1 n

(* booleans *)
let truel = one
let falsel = zero
let andl x y = mul x y
let orl x y = add x y
let notl x = if_zero x (thunk truel) (thunk falsel)
let le x y = notl (sub x y)
let lt x y = notl (le y x)
let eq x y = (andl (le x y) (le y x))

(* lazy conditional *)
let ite b x y = if_zero b y x

(* lists *)
let null = one
let is_null (Ch s) = lt (s (fun x -> x) zero) (s add1 zero)

(* recursion *)
type 'a fix = Fix of ('a fix -> 'a)
(* typed version of the fixpoint combinator y *)
let y t = let p (Fix f) x = t (f (Fix f)) x in p (Fix p)

let fact =
  let t_fact f x =
    (if_zero x (thunk one)
               (Ch (fun _ -> fun _ -> (mul x (f (sub1 x))))))
  in y t_fact

let fib =
  let t_fib f x =
    (ite (lt x two) (thunk x)
    (Ch (fun _ -> fun _ -> (add (f (sub1 x)) (f (sub x two))))))
  in y t_fib

let length =
  let t_length f x =
    ite (is_null x) (thunk zero)
    (Ch (fun _ -> fun _ -> (add1 (f (snd x)))))
  in y t_length
```

```
(* conversions Church -> int and int -> Church
 * for convenience only -- these are not part of the encoding *)

let rec to_church n = if n = 0 then zero else add1 (to_church (n - 1))

let rec from_church (Ch n) =
  try ignore (n (fun _ -> failwith "") zero); 0
  with _ -> 1 + from_church (sub1 (Ch n))
```