L22: Laziness and Streams

Today
1. Lightning tour of OCaml
2. Streams
3. Problem Set 2 revisited, in 82 lines of code

A Note on Programming Languages

This lecture uses OCaml as a representative of functional programming in its natural form, as opposed to the examples we have seen that add functional programming features on top of an object-oriented language. We do not expect you to learn the details of OCaml. Instead, our examples here are meant to serve two purposes:

1. Demonstrate how our “design notations” like datatype definitions can actually be realized in a programming language.

2. Illustrate more of the power of functional programming, hopefully inspiring you to learn more about the subject on your own.

For the curious, here is the OCaml web site URL, for downloading the implementation or reading documentation:

http://caml.inria.fr/ocaml/

Type inference, lists, and options

Compared to the languages we have seen so far, OCaml is distinguished by the power of its type inference, where we may write type-free programs as in Python, but the interpreter infers type information, so that we retain the static typing benefits of Java. For instance, we can define a list:

```ocaml
let my_list = 1 :: 2 :: 3 :: []
```

...and OCaml tells us which type is has come up with:

```ocaml
val my_list : int list = [1; 2; 3]
```

In other words, the interpreter figured out that this is a list of integers.

Type inference can be much more clever. Consider this implementation of the familiar map function:

```ocaml
let rec map f ls =
  match ls with
  | [] -> []
  | x :: ls -> f x :: map f ls
```

The interpreter comes back to us with this type:
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

In other words, map will work correctly for any values of the placeholders 'a and 'b. Given a list with values of some type 'a, we may call map with a function that takes values of type 'a and returns values of type 'b, and then the resulting list contains 'b elements.

Our map function is easy to use on specific inputs:

let my_list' = map (fun x -> x + 1) my_list

We can abbreviate this anonymous function with a convenient notation for treating operators as functions.

let my_list' = map ((+) 1) my_list

Another useful type beside list is option, which captures the idea of a value that may or may not be present. For instance, here is a function to look up the element found in a particular numeric position within a list. We use the constructor None to indicate that the position is out of bounds, and we use the constructor Some to indicate a proper answer. Pattern-matching notation is similar to what we saw last time in Scala, and our definition begins as “let rec” to indicate a recursive function definition.

let rec nth ls n =
    match ls with
    | [] -> None
    | x :: ls ->
        if n = 0 then
            Some x
        else
            nth ls (n-1)

The interpreter infers a type for this function:

val nth : 'a list -> int -> 'a option = <fun>

The 'a placeholder is used to indicate that the result is either None or a Some of a value of the same type as the list elements.

As a quick example of how to manipulate option values, consider this wrapper around nth, which returns a default value instead of None:

let nth_default ls n default =
    match nth ls n with
    None -> default
    | Some v -> v

The inferred type is:

val nth_default : 'a list -> int -> 'a -> 'a = <fun>
Recursive datatypes

In previous lectures, we introduced a “design notation” for recursive datatypes. That notation is actually legal OCaml program syntax. For example, here is how we define the propositional logic formula type that we have used before:

```ocaml
type formula =  
  Var of string  
| Not of formula  
| And of formula * formula  
| Or of formula * formula
```

We write “*” as a sort of Cartesian product, indicating that a datatype constructor takes multiple arguments.

We also used a simple algebraic notation for recursive functions over datatypes. It is not always necessary to translate such notation into Java-like code; OCaml supports it directly. Here is the formula evaluation example we used in a previous lecture.

```ocaml
let rec eval env f =  
  match f with  
  Var x -> env x  
| Not f1 -> not (eval env f1)  
| And (f1, f2) -> eval env f1 && eval env f2  
| Or (f1, f2) -> eval env f1 || eval env f2
```

Its inferred type is:

```ocaml
val eval : (string -> bool) -> formula -> bool = <fun>
```

We represent environments as functions, which is a convenient choice in a functional language.

Introducing streams

With these preliminaries out of the way, we're ready to introduce streams, our main focus in this lecture. Streams are a very effective way of making use of higher-order functions. A stream is a potentially infinite sequence of values. The easiest ways to build streams are the familiar “nil” and “cons” from immutable lists. (We write “unit -> T” for the type of a function with no arguments and return type “T”.)

```ocaml
val nil : unit -> 'a stream  
val cons : 'a -> 'a stream -> 'a stream
```

But the story doesn't stop here. With just these operations, all of our streams would obviously be finite. We would like to be able to build infinite streams, like the stream that is an infinite sequence of zeroes. A naive definition is the following:
let rec allZero () =
    cons 0 (allZero ())

This function isn't very helpful, because a call isZero() loops forever without actually handing us a stream! Surprisingly, though, a minor tweak on this definition can be made to work. The key ingredient is another stream constructor to complement nil and cons:

    val delay : (unit -> 'a stream) -> 'a stream

What is the meaning of a stream built with delay? The idea is that we make a call delay(f), providing a function f that will eventually be called to decide what the stream is. Crucially, this function is not called when the stream is built, but instead only when we finally decide to peek inside the stream. That sort of on-demand computation in programming languages is called laziness.

What is the use of laziness? The example of our allZero stream definition demonstrates. First, let's define a helper function isZero', which is not a stream itself but will return a stream when called.

    let rec allZero' () =
        cons 0 (delay allZero')

Are we in trouble because allZero' always makes a recursive call to itself? No, because the call is hidden inside a delay, so it is not evaluated immediately!

We can finish a definition of the allZero stream with a simple wrapper around allZero':

    let allZero = delay allZero'

How can we test that we got all this right? We will need a key operation for peeking into a stream's contents:

    val force : 'a stream -> ('a * 'a stream) option

In words, when we force a stream, we learn whether it is empty (return value None) or has a first element (return value Some(x, s), where x is the first element, and s is the rest of the stream).

With force, we can implement firstn, a function that extracts the first n elements of a stream, in list form.

    let rec firstn s n =
        if n = 0 then
            []
        else
            match force s with
            None -> []
            | Some (x, s) -> x :: firstn s (n – 1)

Now we can see what's inside our allZero stream (where we write “#” at the beginning of a line of input, to distinguish it from the interpreter's output):
Many other useful stream operations can be derived from just the four primitives above. For instance, we can take the tail of a stream, the result of dropping its first element.

``` Ocaml
let tail s = 
  delay (fun () ->
    match force s with
    None -> failwith "tail: Stream ends!"
  | Some (_, s) -> s)
```

Tail has no effect on the allZero stream, so let's create a more interesting example, with streams of consecutive integers.

``` Ocaml
let rec intsStartingAt n = 
  delay (fun () -> cons n (intsStartingAt (n + 1)))

let naturals = intsStartingAt 0
let positives = tail naturals
```

And some tests for it:

``` Ocaml
# firstn naturals 10;;  
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
# firstn positives 10;;  
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

Tail is a good start, but we can also define some useful higher-order functions that operate on streams. For instance, our old favorite map has a natural stream version.

``` Ocaml
let rec map f s = 
  delay (fun () ->
    match force s with
    None -> nil ()
  | Some (x, s) -> cons (f x) (map f s))
```

Now we have an economical way of creating the allOne stream out of allZero.

``` Ocaml
let allOne = map (fun n -> n + 1) allZero
```

It can also be useful to map over two streams in parallel, using a map2 function:

``` Ocaml
let rec map2 f s1 s2 = 
  delay (fun () ->
```
match force s1, force s2 with
  None, None -> nil ()
| Some (x1, s1), Some (x2, s2) -> cons (f x1 x2) (map2 f s1 s2)
| _, _ -> failwith "List length mismatch"

For instance, we now have another way of defining the positive numbers out of the natural numbers and the allOne stream:

let positivesAgain = map2 (+) naturals allOne

A fresh look at recursive functions, as streams

Here's a familiar face: recursive factorial.

let rec fact n =
  if n <= 1 then
    1
  else
    n * fact (n - 1)

Let's say we want to know all the n! values for n from 0 to 100000. It is easy to define a stream of these answers.

let facts = map fact naturals

However, if we try to firstn this stream up to 100000, we have to wait for a long time. In general, the time is exponential in the size of the stream prefix we choose to force! That's because we reevaluate each factorial value separately.

Instead, let's find a way to reuse n! in calculating (n+1)!, so that we only need linear time to calculate a prefix of facts. Here is where we want the last built-in stream operation in the code for this lecture:

val recursive : ('a stream -> 'a stream) -> 'a stream

This function lets us create a stream that is allowed to refer to itself. To define stream s using some expression e that may mention s, we write:

let s = recursive (fun s -> e)

(The second variable name s needn't be the same as the first; we just write it that way here for clarity, to look more like a normal recursive definition.)

Here's a simple example to redefine allZero more directly.

let allZero = recursive (fun allZero -> cons 0 allZero)

Now we define the stream facts to start with the value 1, followed by the multiplication of facts itself by the positive numbers.

let facts =
  recursive (fun facts -> cons 1 (map2 ( * ) positives facts))
It can take a while to wrap your head around this recursive formulation of facts! It really does support linear-time firstn.

Another fun challenge is writing the Fibonacci function as a recursive stream. Here's the usual definition of Fibonacci:

- \( F(0) = 0 \)
- \( F(1) = 1 \)
- \( F(n+2) = F(n) + F(n+1) \)

Can you see how to reinterpret these equations as a recursive stream?

Here's one solution:

```ocaml
let fibs = recursive (fun fibs -> cons 0 (cons 1 (map2 (+) fibs (tail fibs))))
```

Just as with factorial, the key nice property of this definition is that we can compute the first \( n \) Fibonacci values in linear time, reusing past results to compute new ones.

One other fun example is computing the infinite stream of all the prime numbers. Another standard higher-order function for lists will be useful in its stream form:

```ocaml
let rec filter f s =
  delay (fun () ->
    match force s with
    | None -> nil ()
    | Some (x, s) -> if f x then cons x (filter f s) else filter f s)
```

Now for the prime numbers. The approach we take here is based loosely on the Sieve of Eratosthenes. First we write a helper function sieve, which takes as input a stream of ints, whose first value is guaranteed to be prime; and we output the substream of those numbers that are prime.

```ocaml
let rec sieve ns =
  delay (fun () ->
    match force ns with
    | None -> failwith "Stream ended!"
    | Some (p, xs) -> cons p (sieve (filter (fun n -> n mod p > 0) xs)))
```

Now the primes are just the result of sieving the list of all integers greater than 1.

```ocaml
let primes =
  sieve (intsStartingAt 2)
```

**Problem Set 2 revisited**

As a fun wrap-up to this material, let's think about how Problem Set 2, on symbolic differentiation, can be rewritten to take advantage of streams and other possibilities of OCaml. The code associated with this lecture (available
on Github) contains our complete implementation, as an OCaml source file with just 82 lines of code. We won't reproduce all the code here, but here are some highlights.

We write a lexer that produces tokens in this type:

```ocaml
type token = PLUS | TIMES | NUM of int | VAR of string | LPAREN | RPAREN
```

The lexer itself is a stream-to-stream function with this type:

```ocaml
val lex : char stream -> token stream
```

Next, a parser produces expressions in this type:

```ocaml
type exp = Num of int | Var of string | Plus of exp * exp | Times of exp * exp
```

And the parser is another stream function:

```ocaml
val parse : token stream -> exp stream
```

We have omitted the code for the lexer and parser because they're a bit intricate, but the differentiator itself is trivial, thanks to OCaml's pattern-matching standing in for our earlier uses of the visitor pattern:

```ocaml
let rec differentiate (x : string) (e : exp) : exp =
  match e with
  | Num _ -> Num 0
  | Var s -> Num (if s = x then 1 else 0)
  | Plus (e1, e2) -> Plus (differentiate x e1, differentiate x e2)
  | Times (e1, e2) -> Plus (Times (differentiate x e1, e2),
    Times (e1, differentiate x e2))
```

A similarly straightforward recursive definition gives us

```ocaml
val to_string : exp -> string
```

for formatting expressions as strings for printing.

Now it is trivial to assemble the main loop of our program. We will use a few more library functions. First, a function to apply a function to every element of a stream, purely for its side effects:

```ocaml
val iter : ('a -> unit) -> 'a stream -> unit
```

Next, a stream that gives us all characters typed on the keyboard:

```ocaml
val keyboard : char stream
```

Finally, a function for printing a string on a line by itself:

```ocaml
val print_endline : string -> unit
```

With these tools, we can write the overall loop for PS2 as a simple sequence of operations, like those we built in the map/filter/reduce lecture.

```ocaml
iter print_endline (map to_string
  (map (differentiate "x") (parse (lex keyboard))))
```
The loop behaves properly, waiting for new inputs from the keyboard before outputting new differentiated terms, and finishing when the keyboard input is closed.