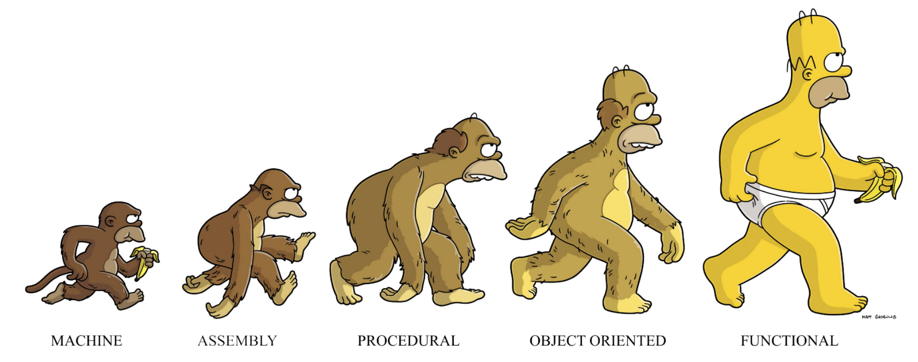
COMP302: Programming Languages and Paradigms

Week 8: Lazy Programming

Prof. Brigitte Pientka <bpientka@cs.mcgill.ca>

School of Computer Science, McGill University



Functional Tidbit:Eager vs Lazy Programming

Eager vs Lazy?

Eager (aka strict) evaluation, aka call-by-value:

- Arguments are evaluated *before* function application,
  e.g. `f e1 e2` is equivalent to

```
let x = e1 in
let y = e2 in
f x y
```

- Variables are bound to *values*,
  e.g. `x` is bound to the result of evaluating `e1`.

Concrete example:

`let x = 3+2 in x * 2`

Evaluate expression `3+2` to the *value* `5`

Associate $x \mapsto 5$ and evaluate `x * 2` to the final value `10`.

Eager vs Lazy?

Another example:

```
let x = horribleComp 345 in 5
```

where `horribleComp` takes several minutes to compute some value, e.g. `777`.

Evaluate `horribleComp 345`.

Associate $x \mapsto 777$.

Evaluate expression `5` to the final value `5` (it's already a value).

Can we skip evaluating `horribleComp 345` and only evaluate when its value is needed?

Lazy: Call By Need

Concrete example:

```
let x = horribleComp (345) in x + x
```

Naive (inefficient) implementation:

1. Associate `x` $\mapsto$ `horribleComp 345`.

$\Longrightarrow$ notice we didn't evaluate `horribleComp 345` yet!

2. Evaluate `x + x` to `horribleComp 345 + horribleComp345`.
3. Evaluate that to a final value.

$\Longrightarrow$ notice `horribleComp 345` was duplicated, and evaluated twice!

Lazy: Call By Need

Concrete example:

```
let x = horribleComp (345) in x + x
```

Naive (inefficient) implementation:

1. Associate `x` $\mapsto$ `horribleComp 345`.

$\implies$ notice we didn't evaluate `horribleComp 345` yet!

2. Evaluate `x + x` to `horribleComp 345 + horribleComp345`.
3. Evaluate that to a final value.

$\implies$ notice `horribleComp 345` was duplicated, and evaluated twice!

Lazy: Call By Need

Concrete example:

```
let x = horribleComp (345) in x + x
```

Efficient implementation:

1. Associate `x` $\mapsto$ `horribleComp 345`.
2. Evaluate `x + x`: *first* evaluate the necessary variable `x`.
   - Evaluate `horribleComp 345` to a value, e.g. `777`.
   - Reassociate `x` $\mapsto$ `777`

   Use the *value* `777` for `x` to evaluate `x + x`.

$\Longrightarrow$ notice `horribleComp 345` is only evaluated once!

Lazy languages like Haskell also avoid evaluating expressions "all the way", e.g. in Haskell `take 2 (map f [1;2;3;4;5;6])` only ends up applying `f` twice!

Wondering about lazy computation

What about unused variables?

```
let x = horribleComp 345 in 5
```

What about side-effects?

Lazy computation is demand-driven

- Harder to reason about.
  (Can't *locally* decide if something is going to be evaluated.)
- Effects must be managed carefully.

+ Can represent infinite data, e.g. the *stream* of all prime numbers.
+ Can represent interactive data, e.g. a stream of web server requests to process.

Eager Computation

+ Easier to reason about
+ Clear when evaluation happens
- May evaluate expressions that are never needed

Finite vs Infinite Data

Finite Data

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

Encodes an *inductive* definition of (finite) lists

- `Nil` is a list of type `'a list`
- If `x` is of type `'a` and `xs` is a list of type `'a list`, then `cons (x,xs)` is a list of type `'a list`.
- Nothing else is a list.

Question	Answer
How do we *build* a list?	By choosing one *constructor*
How do we take apart lists?	By *pattern matching*
How do we reason with lists?	By *induction* on the structure of lists

Infinite Data

Infinite data is not defined by *constructors*...

but instead by the observations we can make on it.

Given an (infinite) stream `1, 2, 3, 4, 5, ...` we can ask for

- the head of the stream obtaining `1`
- the tail of the stream obtaining the stream `2, 3, 4, 5, ...`

Question	Answer
How do we *build* a stream?	By giving *all* observations
How do we take apart a steam?	By choosing *one* observation
How do we reason with streams?	By *co*induction

An observation about the types

We think of *constructors* as having types like

- `Nil : 'a list`
- `Cons : 'a -> 'a list -> 'a list`

We think of *observations* as having types like

- `head : 'a stream -> 'a`
- `tail : 'a stream -> 'a stream`

How to suspend and prevent evaluation of an expression?

```
(** A suspended computation. *)
type 'a susp = Susp of (unit -> 'a)

(* Force evaluation of suspendend computation *)
let force : 'a susp -> 'a =
  fun (Susp f) -> f ()

```

suspend by wrapping it inside a function

```
(* Example of suspending and forcing computation *)
let x = Susp (fun () -> horribleComp 345) in
    force x + force x
```

Note: executed twice

Infinite Data - Streams

```
type 'a str =
  { hd : 'a ;
    tl : 'a str susp
  }
```

Encodes a coinductive definition of infinite streams using the two observations `hd` and `tl`.

- Asking for the head using the observation `hd` returns an element of type `'a`
- Asking for the tail using the observation `tl` returns a suspended stream of type `('a str) susp`.

If you want more elements, you need to ask for more ...

Let's see what this means in practice!

Generating a stream of ones

```
let rec ones =
  {hd = 1 ;
   tl = Susp (fun () -> ones)}

```

How to observe the stream?

```
(* Inspect a stream up to n elements
   take_str : int -> 'a str -> 'a list
*)
let rec take_str n s = match n with
  | 0 -> []
  | n -> s.hd :: take_str (n-1) (force s.tl)
```

Generating a stream of natural numbers!

```
(* val numsFrom : int -> int str *)
let rec numsFrom n =


{hd = ____n______ ;


 tl = ____Susp (fun () -> numsFrom (n+1) )____
}

let nats = numsFrom 0
```

Adding two streams of nats

```
(* addStreams : int str -> int str -> int str  *)
let rec addStreams s1 s2 =


{hd = s1.hd + s2.hd;


 tl = Susp (fun () -> addStreams (force s1.tl) (force s2.tl))
}

```

More Lazy Programming

- Implement a lazy map function on streams
- Implement a lazy zip of streams
- ...

Many programs we wrote about lists have a natural corresponding stream version ...

Stream of Fibonacci Numbers

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$$

The next number is found by adding up the two numbers before it.

n	$Fib(n)$	$+$	$Fib(n+1)$	$Fib(n+2)$
0	$0, 1, \dots$		$1, \dots$	$1, \dots$
1	$1, 1, \dots$		$1, \dots$	$2, \dots$
2	$1, 2, \dots$		$2, \dots$	$3, \dots$
	stream of Fib starting from n		stream of Fib starting from $n+1$	

Stream of Fibonacci Numbers

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...$$

The next number is found by adding up the two numbers before it.

Add up the two streams pointwise

$$\begin{array}{lcccccccccccccc}
\texttt{fibs} & = & 0 & & 1 & & 1 & & 2 & & 3 & & 5 & & \ldots \\
& & + & \searrow & + & \searrow & + & \searrow & + & \searrow & + & \searrow & & & \\
\texttt{fibs'} & = & 1 & & 1 & & 2 & & 3 & & 5 & & 8 & & \ldots
\end{array}$$

$$
\begin{array}{lcccccccccccccc}
\texttt{fibs} & = & 0 & & 1 & & 1 & & 2 & & 3 & & 5 & & \ldots \\
 & & & + & \searrow & + & \searrow & + & \searrow & + & \searrow & + & \searrow & & \\
\texttt{fibs'} & = & 1 & & 1 & & 2 & & 3 & & 5 & & 8 & & \ldots
\end{array}
$$

```
let rec fibs =

{ hd = 0 ;

  tl = Susp (fun () -> fibs' ) }

and fibs' =

{hd = 1 ;

 tl = Susp (fun () -> addStreams fibs fibs' ) }
```

I want to see the stream of Fibonacci numbers ...

Try it out and observe the first 10 elements of the stream!

```
# take_str 10 fibs;;
- : int list = [0; 1; 1; 2; 3; 5; 8; 13; 21; 34]
```

What about possibly infinite lists?

- Streams are *always* infinite.
- What if we could have maybe infinite, maybe finite lists?

```
type 'a lazy_list =
  { hd: 'a
  ; tl : 'a fin_list susp
  }

and 'a fin_list = Empty | NonEmpty of 'a lazy_list
```

coinductive

inductive

Idea: interleave an *inductive* type with a *coinductive* type.

- `'a fin_list` captures the possibility for the list to end.
- `'a lazy_list` (with its suspended computation) captures the possibility for the list to be infinite.

Lazy List of Natural Numbers

```
(*
val natsFrom : int -> int lazy_list =
val natsFrom' : int -> int fin_list =
*)
let rec natsFrom n =
  { hd = n ;
    tl = Susp (fun () -> natsFrom' (n-1)) }

and natsFrom' n = if n < 0 then Empty
                  else NonEmpty (natsFrom n)
```

produce all numbers from n to 0

Revisiting Map

```
(* val map : ('a -> 'b) -> 'a lazy_list -> 'b lazy_list
   val map' : ('a -> 'b) -> 'a fin_list -> 'b fin_list
*)
let rec map f s =
{ hd = f s.hd ;
  tl = Susp (fun () -> map' f (force s.tl))
}

and map' f xs = match xs with
  | Empty -> Empty
  | NonEmpty xs -> NonEmpty (map f xs)
```

Take-Away

- Functions can be used to suspend computation
- Model infinite data such as streams using functions to delay computation and records to model the observations we can make about the data