[CSE301 / Lecture 5] Laziness and infinite objects

Noam Zeilberger

Ecole Polytechnique

2 October 2024

What is laziness?

The dominant state of most students.

Also, an **evaluation strategy** used by Haskell.

Idea: only evaluate something if it is needed to compute the result of the overall computation, and once you've evaluated something, don't evaluate it again.

You can try this on the lab machines...

In ghci:

```
ghci> :set +m
ghci> ack m n = if m == 0 then n+1
ghci|           else if n == 0 then ack (m-1) 1
ghci|           else ack (m-1) (ack m (n-1))
ghci> let x = ack 4 3 in 1+1
2
```

In ocaml:

```
# let rec ack m n = if m == 0 then n+1
                    else if n == 0 then ack (m-1) 1
                    else ack (m-1) (ack m (n-1)) ;;
val ack : int -> int -> int = <fun>
# let x = ack 4 3 in 1+1 ;;
Warning 26: unused variable x.
[...this will take a while...]
```

Laziness in Haskell

In Haskell, evaluation is lazy by default, for better or worse:

- Often can be used to turn seemingly naive mathematical formulas into efficient algorithms.
- Allows for elegant encodings of infinite objects

But...

- It makes it harder to write a compiler
- Often much harder to reason about performance

Example: the Fibonacci sequence

The following is valid Haskell code, defining the infinite sequence of Fibonacci numbers.

$$\mathit{fibseq} = 0 : 1 : \mathit{zipWith}\ (+)\ \mathit{fibseq}\ (\mathit{tail}\ \mathit{fibseq})$$

We can use it to give another definition of the function *fib*:

$$\mathit{fib}\ n = \mathit{fibseq}\ !!\ n$$

This runs in linear time, and remembers (memoizes) its results!

Plan for today

We will try to cover these topics:

1. Evaluation
2. Evaluation strategies for functional languages
3. Laziness and infinite objects
4. Computational duality
5. Overcoming laziness

Evaluation

Recall that an **expression** denotes a computation towards a **value**. The process of computing that value is called **evaluation**.

Evaluation may be visualized as a series of reductions[1] from one expression to another expression, ending in a value, e.g.:

$$\begin{aligned}(1+2)*3 &\rightarrow 3*3 \\ &\rightarrow 9\end{aligned}$$

[1]In practice, this is not the way evaluation is implemented. Rather, a program may be compiled and executed as machine code, or alternatively evaluated by an interpreter using an *abstract machine*. Nevertheless, thinking of evaluation of a functional program as a series of reductions is a good mental model to have when reasoning about its behavior, to a first approximation.

Evaluation

In general, an expression may also produce some side-effects along the way towards computing a value (even in Haskell).

$$(\mathit{putStrLn}\ \texttt{"hi"} \gg \mathit{return}\ ((1+2)*3)) \underset{\substack{\wr \\ \downarrow \\ \mathit{hi}}}{\longrightarrow} (1+2)*3 \twoheadrightarrow 9$$

So the general shape of evaluation looks like this:

$$\mathit{expression} \underset{\substack{\wr \\ \downarrow \\ \mathit{side\text{-}effects}}}{\xrightarrow{\qquad\qquad}\!\!\!\twoheadrightarrow} \mathit{value}$$

Evaluation

To make evaluation precise, we need to explain:

- What counts as a value
- How to perform reductions (and execute side-effects, if any)
- *Where* to perform reductions

Such an explanation is called an **evaluation strategy**.

Evaluation in pure λ-calculus (aka normalization)

One rule of reduction (β):

$$(\lambda x.e_1)(e_2) \to e_1[e_2/x]$$

Can be performed *anywhere* (i.e., on any matching “redex”).

Value = expression with no redex

The order we perform β-reductions does not matter for the final value (Church-Rosser Theorem), but might make a difference to how quickly we reach a value, and even to whether we reach one.

Evaluation in pure λ-calculus (aka normalization)

A term with two β-redices:

$$\underline{(\lambda x.\lambda y.y)(\underline{(\lambda z.zz)(\lambda z.zz)}_2)}_1$$

Two very different reduction paths:

$$\begin{array}{ccc} (\lambda x.\lambda y.y)((\lambda z.zz)(\lambda z.zz)) & \xrightarrow{1} & \lambda y.y \\ \downarrow_2 & & \\ (\lambda x.\lambda y.y)((\lambda z.zz)(\lambda z.zz)) & & \\ \downarrow_2 & & \\ \vdots & & \end{array}$$

Evaluation in pure λ-calculus (aka normalization)

There is a deterministic evaluation strategy that always succeeds to find a β-normal form, if it exists: pick the leftmost redex which is not contained in another redex ("leftmost outermost" reduction).

But this is *not* the evaluation strategy used in Haskell or OCaml...

Call-by-value[2]

In **call-by-value** (CBV) evaluation, the argument to a function is always reduced to a value before calling the function.

Now, a value can be *any* function (e.g., may contain β-redices), or a constructor applied to some *values.*

[2]Used by OCaml, Python, C, Java, and many other languages.

Call-by-value

For example, let $\mathit{sqr}\ x = x * x$ and $\mathit{const0}\ x = 0$

Under CBV evaluation:

$$\mathit{sqr}\ (1+2) \rightarrow \mathit{sqr}\ 3 \rightarrow 3 * 3 \rightarrow 9$$

$$\mathit{const0}\ (\mathit{sqr}\ 3) \rightarrow \mathit{const0}\ (3 * 3) \rightarrow \mathit{const0}\ 9 \rightarrow 0$$

Call-by-name[3]

In **call-by-name** (CBN) evaluation, the argument to a function is passed as an unevaluated expression ("by name").

A value is any function, or a constructor applied to *expressions.*

Under CBN evaluation:

$$\textit{sqr}\ (1+2) \to (1+2) * (1+2) \to 3 * (1+2) \to 3 * 3 \to 9$$

$$\textit{const0}\ (\textit{sqr}\ 3) \to 0$$

[3] Of historical interest (e.g., Algol 60), but *not* used by Haskell...

CBV vs CBN

"CBV is better": avoid re-evaluating the argument to a function.

"CBN is better": avoid evaluating an argument that is unneeded.

How do you decide?


Why don't we have both?

Call-by-need[4]

In **call-by-need** evaluation, the argument to a function is only evaluated when it is needed, and then stored for later reuse.

Call-by-need is also called *lazy evaluation*.

Roughly, it is implemented by giving names to intermediate computations ("thunks"), and evaluating them on demand.

[4] Used by Haskell.

Call-by-need

$$
\begin{aligned}
\mathit{sqr}\ (1+2) &\rightarrow \textbf{let}\ x = 1+2\ \textbf{in}\ \mathit{sqr}\ x && \text{[introduce thunk]} \\
&\rightarrow \textbf{let}\ x = 1+2\ \textbf{in}\ x * x && \text{[apply function]} \\
&\rightarrow \textbf{let}\ x = 3\ \textbf{in}\ x * x && \text{[evaluate thunk]} \\
&\rightarrow \textbf{let}\ x = 3\ \textbf{in}\ 3 * 3 && \text{[fetch value]} \\
&\rightarrow \textbf{let}\ x = 3\ \textbf{in}\ 9 && \text{[evaluate expression]} \\
&\rightarrow 9 && \text{[garbage collect]}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{const0}\ (\mathit{sqr}\ 3) &\rightarrow \textbf{let}\ x = \mathit{sqr}\ 3\ \textbf{in}\ \mathit{const0}\ x && \text{[introduce thunk]} \\
&\rightarrow \textbf{let}\ x = \mathit{sqr}\ 3\ \textbf{in}\ 0 && \text{[apply function]} \\
&\rightarrow 0 && \text{[garbage collect]}
\end{aligned}
$$

The cost of laziness

Although call-by-need is “better” than CBV or CBN in the sense of performing less evaluation, it comes at a cost:

- The computational cost (time + space) of managing thunks
- The engineering cost of implementing it correctly in a compiler
- The mental cost of reasoning about program performance

Nevertheless, it can be used to write some pretty code!!

Understanding Fibonacci

Recall the one-liner:

$$fibseq = 0 : 1 : zipWith\ (+)\ fibseq\ (tail\ fibseq)$$

Why does this work?

Understanding Fibonacci

We can use the definition

$$\textit{fibseq} = 0 : 1 : \textit{zipWith}\ (+)\ \textit{fibseq}\ (\textit{tail fibseq})$$

to build up a table of values...

fibseq	0	1						
tail fibseq	1							
tail (*tail fibseq*)								

Understanding Fibonacci

We can use the definition

$$fibseq = 0 : 1 : zipWith\ (+)\ fibseq\ (tail\ fibseq)$$

to build up a table of values...

fibseq	0	1						
tail fibseq	1							
tail (tail fibseq)	1							

Understanding Fibonacci

We can use the definition

$$fibseq = 0 : 1 : zipWith\ (+)\ fibseq\ (tail\ fibseq)$$

to build up a table of values...

fibseq	0	1	1					
tail fibseq	1	1						
tail (tail fibseq)	1							

Understanding Fibonacci

We can use the definition

$$fibseq = 0 : 1 : zipWith\ (+)\ fibseq\ (tail\ fibseq)$$

to build up a table of values...

fibseq	0	1	1					
tail fibseq	1	1						
tail (tail fibseq)	1	2						

Understanding Fibonacci

We can use the definition

$$fibseq = 0 : 1 : zipWith\ (+)\ fibseq\ (tail\ fibseq)$$

to build up a table of values...

fibseq	0	1	1	2				
tail fibseq	1	1	2					
tail (*tail fibseq*)	1	2						

Understanding Fibonacci

We can use the definition

$$fibseq = 0 : 1 : zipWith\ (+)\ fibseq\ (tail\ fibseq)$$

to build up a table of values...

fibseq	0	1	1	2	3			
tail fibseq	1	1	2	3				
tail (tail fibseq)	1	2	3					

Understanding Fibonacci

We can use the definition

$$\mathit{fibseq} = 0 : 1 : \mathit{zipWith}\ (+)\ \mathit{fibseq}\ (\mathit{tail}\ \mathit{fibseq})$$

to build up a table of values...

fibseq	0	1	1	2	3	5	8	···
tail fibseq	1	1	2	3	5	8		
tail (*tail fibseq*)	1	2	3	5	8			

Now in GHCi

Using the ":sprint" command to inspect a lazy value...

```
ghci> :sprint fibseq
fibseq = _
ghci> fib 3
2
ghci> :sprint fibseq
fibseq = 0 : 1 : 1 : 2 : _
ghci> fib 7
13
ghci> :sprint fibseq
fibseq = 0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : _
```

Even and odd numbers, v1

$$
\begin{aligned}
&\mathit{nats}, \mathit{evens}, \mathit{odds} :: [\,\mathit{Integer}\,] \\
&\mathit{nats} = [0\,..] \\
&\mathit{evens} = \mathit{map}\ (*2)\ \mathit{nats} \\
&\mathit{odds} = \mathit{map}\ (+1)\ \mathit{evens}
\end{aligned}
$$

Even and odd numbers, v1

```
ghci> :sprint nats
nats = _
ghci> :sprint odds
odds = _
ghci> take 5 odds
[1,3,5,7,9]
ghci> :sprint nats
nats = 0 : 1 : 2 : 3 : 4 : _
```

Even and odd numbers, v2

$$
\begin{aligned}
&\mathit{nats}', \mathit{evens}', \mathit{odds}' :: [\mathit{Integer}] \\
&\mathit{evens}' = 0 : \mathit{map}\ (+1)\ \mathit{odds}' \\
&\mathit{odds}' = \mathit{map}\ (+1)\ \mathit{evens}' \\
&\mathit{nats}'\ = \mathit{interleave}\ \mathit{evens}'\ \mathit{odds}' \\
&\quad \textbf{where}\ \mathit{interleave}\ (x : \mathit{xs})\ \mathit{ys} = x : \mathit{interleave}\ \mathit{ys}\ \mathit{xs}
\end{aligned}
$$

Even and odd numbers, v2

```
ghci> :sprint nats'
nats' = _
ghci> take 5 nats'
[0,1,2,3,4]
ghci> :sprint evens'
evens' = 0 : 2 : 4 : _
ghci> :sprint odds'
odds' = 1 : 3 : _
```

Even and odd numbers, v3

$$
\begin{aligned}
&everyOther :: [a] \to [a] \\
&everyOther\ (x : y : xs) = x : everyOther\ xs \\
&evens'', odds'' :: [Integer] \\
&evens'' = everyOther\ nats \\
&odds'' = everyOther\ (tail\ nats)
\end{aligned}
$$

Even and odd numbers, v3

```
ghci> :sprint nats
nats = 0 : 1 : 2 : 3 : 4 : _
ghci> take 5 odds''
[1,3,5,7,9]
ghci> :sprint nats
nats = 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : 10 : _
```

Another version of Fibonacci

Another one-liner:

$$fibseq = map\ fst\ \$\ iterate\ (\backslash(a, b) \rightarrow (b, a + b))\ (0, 1)$$

where *iterate* is defined in the Prelude:

$$iterate :: (a \rightarrow a) \rightarrow a \rightarrow [a]$$
$$iterate\ f\ x = x : iterate\ f\ (f\ x)$$

i.e., *iterate f x* (lazily) builds the infinite list $[x, f\ x, f\ (f\ x), ...]$.

Computational duality

Back in Lecture 1, we saw how to define data types by their constructors, and how to define functions over such types by pattern-matching against those possible constructors.

But there is also a dual way of defining a type by its *destructors*.

A value of such a type[5] can then be defined by matching against those possible destructors.

Category theory is good at making such definitions...

[5] Sometimes called a "codata" type or a "negative" type.

Products, in category theory

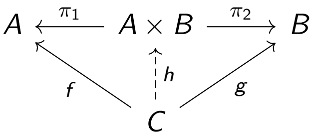
The <u>product</u> of objects A and B is an object $A \times B$ with arrows

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$$

such that for any other pair of arrows

$$A \xleftarrow{f} C \xrightarrow{g} B$$

there is a unique arrow making the diagram below "commute":



Translating the category theory to Haskell?

Given $f :: c \to a$ and $g :: c \to b$, we could hope to define

$$\begin{aligned} &h :: c \to (a, b) \\ &\mathit{fst}\ (h\ x) = f\ x \\ &\mathit{snd}\ (h\ x) = g\ x \end{aligned}$$

but unfortunately this is not (currently) legal Haskell syntax.[6]

Still, this "observational" perspective is good to keep in mind.

---

[6] Although it should be! For example, Agda supports copattern-matching. For more on the theoretical foundations for copattern-matching, see the paper "Copatterns: Programming Infinite Structures by Observations" by Abel et al.

Redefining lists, observationally

We can think of an infinite list as defined by its behavior against the destructors $\mathit{head} :: [a] \rightarrow a$ and $\mathit{tail} :: [a] \rightarrow [a]$.

For example, the following (legal Haskell) definition

$$\mathit{ones} = 1 : \mathit{ones}$$

can be thought of as defining a value by the equations

$$\mathit{head}\ \mathit{ones} = 1$$
$$\mathit{tail}\ \mathit{ones} = \mathit{ones}$$

Redefining lists, observationally

The reason we can manipulate infinite values in computations is because any given *observation* is finite.

$$\mathit{head}\ \mathit{ones} = 1$$

$$\mathit{head}\ (\mathit{tail}\ (\mathit{tail}\ \mathit{ones})) = \mathit{head}\ (\mathit{tail}\ \mathit{ones}) = \mathit{head}\ \mathit{ones} = 1$$

Record syntax

Although Haskell does not have copattern-matching, it does have record types equipped with named fields.

$$\textbf{data}\ \mathit{Stream}\ a = \mathit{Stream}\ \{\ \mathit{hd} :: a, \mathit{tl} :: \mathit{Stream}\ a\ \}$$

$$\mathit{oneS} :: \mathit{Stream}\ \mathit{Integer}$$
$$\mathit{oneS} = \mathit{Stream}\ \{\ \mathit{hd} = 1, \mathit{tl} = \mathit{oneS}\ \}$$

```
ghci> hd (tl (tl oneS))
1
```

Overcoming laziness

Sometimes laziness gets in the way in Haskell. There are a few techniques for working around it:

- the *seq* operator to force evaluation
- strictness annotations for non-lazy data types
- monads (or CPS) to ensure lazy computations happen in a certain order

But first a puzzle...

Suppose we define $minimum = head \circ sort$.

What is the complexity of computing $minimum\ xs$?

The *seq* operator

Takes two arguments and returns the second

$$seq :: a \to b \to b$$

but forces evaluation of the first argument.

```
ghci> seq "hello" 42
42
ghci> seq (ack 4 3) (1+1)
  C-c C-cInterrupted.
```

Strictness annotations

```
data StrictList a = Nil | Cons !a !(StrictList a)
  deriving (Show, Eq)

toSL :: [a] → StrictList a
toSL [] = Nil
toSL (x : xs) = Cons x (toSL xs)

nullSL :: StrictList a → Bool
nullSL Nil = True
nullSL _ = False
```

Strictness annotations

```
ghci> xs = take 5 fibseq
ghci> null xs
False
ghci> :sprint xs
xs = 0 : _
ghci> ys = toSL (take 5 fibseq)
ghci> nullSL ys
False
ghci> :sprint ys
ys = Cons 0 (Cons 1 (Cons 1 (Cons 2 (Cons 3 Nil))))
ghci> toSL fibseq
  C-c C-cInterrupted.
```

Summary

Key points from today:

- An *evaluation strategy* is a plan for reducing expressions to values (and performing any side-effects present)
- Different languages use different evaluation strategies
- Haskell uses *call-by-need* (a.k.a. "lazy") evaluation, meaning function arguments are only evaluated when they are needed
- Laziness can sometimes yield significant performance gains, and enables finite representations of infinite values
- But laziness comes at a cost (compiler + runtime + brain)