CIS 1904: Haskell

Lazy Evaluation

Logistics

- HW05 available this evening
- Next week: Typeclasses
- Two weeks from now (3/6): Review
  - Please complete (anonymous) Canvas quiz on what topics to review

What is Lazy Evaluation?

Main idea: Haskell does not evaluate function arguments until needed

- In Haskell, arguments are implicitly boxed up inside *thunks*
- Unpackaging and evaluating an argument to inspect it is called *forcing* it

What is Lazy Evaluation?

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

lastOrDefault :: List a -> a -> a
lastOrDefault [] def = def
lastOrDefault [h] _ = h
lastOrDefault (h : tl) _ = lastOrDefault tl
```

What is Lazy Evaluation?

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

addLastOrDefault 2 0 (sort [4,1,3])
```

What is Lazy Evaluation?

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

addLastOrDefault 2 0 (sort [4,1,3])
```

The Alternative: Strict Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

addLastOrDefault 2 0 (sort [4,1,3])
```

The Alternative: Strict Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

addLastOrDefault 2 0 ([1,3,4])
```

The Alternative: Strict Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

2 + lastOrDefault 0 ([1,3,4])
```

The Alternative: Strict Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

2 + 4
```

The Alternative: Strict Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs


6
```

The Alternative: Strict Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

6
```

This is what you see in most programming languages (OCaml, Java, etc.)!

Lazy Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

addLastOrDefault 2 0 (sort [4,1,3])
```

Lazy Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

addLastOrDefault 2 0 (sort [4,1,3])
```

Main idea: a function argument does not get evaluated until it is needed

Lazy Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

addLastOrDefault 2 0 (sort [4,1,3])
```

Main idea: a function argument does not get evaluated until it is needed

Lazy Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

2 + lastOrDefault 0 (sort [4,1,3])
```

Main idea: a function argument does not get evaluated until it is needed

Lazy Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

2 + lastOrDefault 0 (sort [4,1,3])
```

Main idea: a function argument does not get evaluated until it is needed

Lazy Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

2 + lastOrDefault 0 ([1,3,4])
```

Main idea: a function argument does not get evaluated until it is needed

Lazy Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

2 + 4
```

Main idea: a function argument does not get evaluated until it is needed

Lazy Evaluation

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

6
```

Main idea: a function argument does not get evaluated until it is needed

Why is this useful?

```
addTwo :: Int -> Int -> Int
addTwo x y = x + 2
```

Why is this useful?

```
addTwo :: Int -> Int -> Int
addTwo x y = x + 2
```

```
addTwo 5 (29^210)
```

Why is this useful?

```
addTwo :: Int -> Int -> Int
addTwo x y = x + 2
```

```
addTwo 5 (29^210)
```

Why is this useful?

```
addTwo :: Int -> Int -> Int
addTwo x y = x + 2
```

```
addTwo 5 (29^210)
    → addTwo 5 (29 * 29 * 29 … * 29)
```

Why is this useful?

```
addTwo :: Int -> Int -> Int
addTwo x y = x + 2
```

```
addTwo 5 (29^210)
    → addTwo 5 (29 * 29 * 29 … * 29)
    → addTwo 5 (841 * 29 * … * 29)
```

Why is this useful?

```
addTwo :: Int -> Int -> Int
addTwo x y = x + 2
```

```
addTwo 5 (29^210)
    → addTwo 5 (29 * 29 * 29 … * 29)
    → addTwo 5 (841 * 29 * … * 29)
    → addTwo 5 (24389 * … * 29)
```

Why is this useful?

```
addTwo :: Int -> Int -> Int
addTwo x y = x + 2
```

```
addTwo 5 (29^210)
    → addTwo 5 (29 * 29 * 29 … * 29)
    → addTwo 5 (841 * 29 * … * 29)
    → addTwo 5 (24389 * … * 29)
    …
    → addTwo 5 (12693446555…)
```

Why is this useful?

```
addTwo :: Int -> Int -> Int
addTwo x y = x + 2
```

```
addTwo 5 (29^210)
    → addTwo 5 (29 * 29 * 29 … * 29)
    → addTwo 5 (841 * 29 * … * 29)
    → addTwo 5 (24389 * … * 29)
    …
    → addTwo 5 (12693446555…)
    → 5 + 2
```

Why is this useful?

```
addTwo :: Int -> Int -> Int
addTwo x y = x + 2
```

```
addTwo 5 (29^210)
    → addTwo 5 (29 * 29 * 29 … * 29)
    → addTwo 5 (841 * 29 * … * 29)
    → addTwo 5 (24389 * … * 29)
    …
    → addTwo 5 (12693446555…)
    → 5 + 2
    → 7
```

Why is this useful?

```
addTwo :: Int -> Int -> Int
addTwo x y = x + 2
```

```
addTwo 5 (29^210)
    → addTwo 5 (29 * 29 * 29 … * 29)
    → addTwo 5 (841 * 29 * … * 29)
    → addTwo 5 (24389 * … * 29)
    …
    → addTwo 5 (12693446555…)
    → 5 + 2
    → 7
```

Why is this useful?

```haskell
addTwo :: Int -> Int -> Int
addTwo x y = x + 2
```

```haskell
addTwo 5 (29^210)
```

Why is this useful?

```
addTwo :: Int -> Int -> Int
addTwo x y = x + 2
```

```
addTwo 5 (29^210)
    → 5 + 2
    → 7
```

Why is this useful?

```
addTwoSometimes :: Int -> Int -> Bool -> Int
addTwoSometimes x y b =
    if b then
        x + y
    else
        x + 2
```

Why is this useful?

```
addTwoSometimes :: Int -> Int -> Bool -> Int
addTwoSometimes x y b =
    if b then
        x + y
    else
        x + 2
```

```
addTwoSometimes 5 (29^210) complexBooleanFormula
```

Why is this useful?

```
(&&) :: Bool -> Bool -> Bool
True && x = x
False && _ = False
```

What’s the catch?

```
sum_debug :: List Int -> Int
sum_debug xs =
    print “Running sum!”
    sum xs


app_debug :: List a -> List a -> List a
app_debug xs ys =
    print “Running append!”
    xs ++ ys
```

Note: this is not valid Haskell

What’s the catch?

```
sum_debug :: List Int -> Int
sum_debug xs =
    print “Running sum!”
    sum xs


app_debug :: List a -> List a -> List a
app_debug xs ys =
    print “Running append!”
    xs ++ ys
```

```
> sum_debug (app_debug [6] [4])
```

Note: this is not valid Haskell

What’s the catch?

```
sum_debug :: List Int -> Int
sum_debug xs =
    print “Running sum!”
    sum xs
```

```
> sum_debug (app_debug [6] [4])

Running sum!
Running append!
10
```

```
app_debug :: List a -> List a -> List a
app_debug xs ys =
    print “Running append!”
    xs ++ ys
```

Note: this is not valid Haskell

What’s the catch?

Laziness makes it hard to reason about *side effects*.

Side effect: any way the program interacts with the outside world

- Printing
- Reading from memory
- Crashing

Why is this useful?

```
addTwo :: Int -> Int -> Int
addTwo x y = x + 2



exp_debug :: Int -> Int -> Int
exp a b =
    print “Running exp!”
    a^b
```

```
> addTwo 5 (exp 29 210)
```

Why is this useful?

```
addTwo :: Int -> Int -> Int
addTwo x y = x + 2


exp_debug :: Int -> Int -> Int
exp a b =
    print “Running exp!”
    a^b
```

```
> addTwo 5 (exp 29 210)

7
```

What’s the catch?

Laziness makes it hard to reason about *side effects*.

Side effect: any way the program interacts with the outside world

- Printing
- Reading from memory
- Crashing

Haskell is (mostly) a *pure* language — it has (mostly) no side effects!

Q: When do lazy expressions get evaluated?

Q: When do lazy expressions get evaluated?

A: When we pattern match on them.

What is Lazy Evaluation?

Main idea: a function call does not get evaluated until its output is needed

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

lastOrDefault :: List a -> a -> a
lastOrDefault [] def = def
lastOrDefault [h] _ = h
lastOrDefault (h : tl) _ = lastOrDefault tl
```

What is Lazy Evaluation?

Main idea: a function call does not get evaluated until its output is needed

```
addLastOrDefault :: Int -> Int -> List Int -> Int
addLastOrDefault i def xs = i + lastOrDefault def xs

lastOrDefault :: List a -> a -> a
lastOrDefault [] def = def
lastOrDefault [h] _ = h
lastOrDefault (h : tl) _ = lastOrDefault tl
```

We need to evaluate to know which case we're in!

When do lazy expressions get evaluated?

```
foo :: Maybe a -> [Maybe a]
foo m = [m, m]
```

Will foo1’s argument be evaluated during its execution?

When do lazy expressions get evaluated?

```
foo :: Maybe a -> [Maybe a]
foo m = [m, m]
```

Will foo1's argument be evaluated during its execution?

No, we do not need to pattern match on `m` to place it in a list.

When do lazy expressions get evaluated?

```
foo2 :: Maybe a -> [a]
foo2 Nothing  = []
foo2 (Just x) = [x]
```

Will foo2's argument be evaluated during its execution?

When do lazy expressions get evaluated?

```
foo2 :: Maybe a -> [a]
foo2 Nothing  = []
foo2 (Just x) = [x]
```

Will foo2's argument be evaluated during its execution?

Partially – we need to know if its argument is `Nothing` or `Just`, but we do not need to evaluate `x`.

When do lazy expressions get evaluated?

```
foo2 :: Maybe a -> [a]
foo2 Nothing  = []
foo2 (Just x) = [x]


foo3 :: Int -> [Int]
foo3 x = foo2 (Just x^210)
```

Will foo3's argument be evaluated?

No – we only need to pattern match to know if we are in the `Nothing` case or the `Just` case. We do not need to match on the integer.

Strict Evaluation: !

- We can use ! on arguments (and sometimes operators) to force Haskell to evaluate them strictly

```
addTwo 5 !(29^210)
= addTwo 5 (29 * 29 * … * 29)
= addTwo 5 12693446555…
= 5 + 2
= 7
```

Terminology

- Call-by-value ~ strict
- Call-by-name ~ lazy
  - Call-by-need ~ lazy with caching (what Haskell does!)

```
triple :: Int -> Int
triple x = x + x + x


triple (29^210)
= 29^210 + 29^210 + 29^210
= 12693446555… + 12693446555… + 12693446555…
```

Infinite Data Structures

Laziness lets us operate on infinite data structures

```
natsSimple :: [Int]
natsSimple = 1 : map (1 +) natsSimple

1 : map (1+) natsSimple
1 : map (1+) (1 : map (1+) natsSimple)
1 : map (1+) (1 : map (1+) (1 : map (1+) natsSimple))
1 : 1+1 : map (1+) (map (1+) (1 : map (1+) natsSimple))
1 : 1+1 : map (1+) (1+1 : map (1+) (map (1+) natsSimple))
1 : 1+1 : 1+1+1 : map (1+) (map (1+) (map (1+) natsSimple))
1 :  2  :   3   : map (1+) (map (1+) (map (1+) natsSimple))
```