Lazy Evaluation
Edward Z. Yang
Motivation
- What is lazy evaluation?
- Why does laziness matter?

Technical meat:
- "Graph" evaluation
- Working knowledge of stream combinators

Beyond Haskell:
- How it's implemented
- Generators
What is lazy evaluation?

let \( f \ x \ y = x+2 \) in

\( f \ 5 \ (29^{35792}) \)
What is strict evaluation?

let \( f \) \( x \ y \) = \( x \ + \ 2 \) in

\[ f \ 5 \ \ (29^{35792}) \]

\textit{evaluate me}
What is strict evaluation?

\[
\text{let } f x y = x + 2 \text{ in }
\]

\[
f 5 \ (29^{35792})
\]
What is strict evaluation?

\[
\text{let } f x y = x + 2 \text{ in }
\]

\[
f \ 5 \ 1409745767702881930
\]

evaluate f
What is lazy evaluation?

\[ \text{let } f x y = x + 2 \text{ in} \]

\[ f \ 5 \ (29^{35792}) \]

\[ \uparrow \]

evaluate \( f \)
What is lazy evaluation?

let \( f \ x \ y = x + 2 \) in

\[
f \begin{bmatrix} 5 \end{bmatrix} (29 \uparrow 35792)
\]

↓ suspended as \textbf{thunks}
What is lazy evaluation?

- Don’t evaluate expressions until they are needed
  ~ we’ll make this precise today

- Evaluate an expression once, then memoize the result for later

Recall call-by-name:

\[
(\lambda x. x + x) \ (29^{35792}) \quad \rightarrow \quad 29^{35792} + 29^{35792}
\]
Why does lazy evaluation matter?

Streams

Generators

0 → 0 → 0 → 0 → ...

Monotonicity

1 ≤ 1:1 ≤ 1:2:1 ≤ ...

Functional programming

DSLs
Compositionality
Cyclic data structures
Why does lazy evaluation matter? Streams

- Lots of data is too big to fit in memory: want to process as you go

- Streams are a lazy data structure

Payload 1 ➔ Payload 2 ➔ thunk rest of the stream
Why does lazy evaluation matter? Monotonicity

How to prevent data races between thread 1 & 2?

(I said FP would be good for concurrency)
Why does lazy evaluation matter? *Monotonicity*

It's **impossible** to race with thunks *(Correct-by-construction abstraction)*

(I said FP would be good for concurrency)
Why does lazy evaluation matter? FP

DSLs

```java
many(
  function() {
    string();
    char('!',);
    3);
  };
```

vs.

```java
many (string >> char '!',)
```

DSL friendly languages have compact syntax for closures
Why does lazy evaluation matter?  FP

Compositionality

\[
\text{any} : (a \to \text{Bool}) \to [a] \to \text{Bool}
\]

\[
\text{any } p = \text{or } \cdot \text{map } p
\]

\[
[a] \to [\text{Bool}]
\]

\[
[\text{Bool}] \to \text{Bool}
\]

This is lazy! As soon as we find the first `a` that satisfies the predicate, we stop
Why does lazy evaluation matter?  FP

Compositionality

\[ \text{any} :: (a \to \text{Bool}) \to [a] \to \text{Bool} \]
\[ \text{any } p = \text{or. } \text{map } p \]

Strict language would be obligated to fully evaluate map \( p \) before continuing
Why does lazy evaluation matter? FP

Cyclic data structures

Graphs are difficult in a pure language
Why does lazy evaluation matter?  FP

Cyclic data structures

Graphs are difficult in a pure language

let a = Node []
Why does lazy evaluation matter?  FP

Cyclic data structures

Graphs are difficult in a pure language

```ml
let a = Node []
let b = Node [a, ???]
```
Why does lazy evaluation matter?  FP

Cyclic data structures

Graphs are difficult in a pure language

```plaintext
let a = Node []
let c = Node [???]
let b = Node [a, c]
```

normally, you’d use mutation to fill in edges after b is allocated
Why does lazy evaluation matter? FP

Cyclic data structures

Graphs are difficult in a pure language

```
let a = Node []
c = Node [b]
b = Node [a, c]
```

![Diagram of cyclic data structure]

with laziness, no problem!

(of course, mutating a graph like this is another matter...)
TIME FOR SOME DETAILS
Don’t evaluate expressions until they are needed.

When is the argument of a function needed?
\[ f_1 :: \text{Maybe } a \rightarrow [\text{Maybe } a] \]
\[ f_1 \, m = [m, m] \]

\[ f_2 :: \text{Maybe } a \rightarrow [a] \]
\[ f_2 \, \text{Nothing} = [] \]
\[ f_2 \, (\text{Just } x) = [x] \]

\text{print} \ (\text{null} \ (f_1 \ f_2 \, m))
\( f_1 : \text{Maybe} \, a \rightarrow [\text{Maybe} \, a] \)
\( f_1 \, m = [m, m] \)

We "used" the argument, but we don't care what it actually is.

\( f_2 : \text{Maybe} \, a \rightarrow [a] \)
\( f_2 \, \text{Nothing} = [] \)
\( f_2 \, (\text{Just} \, x) = [x] \)

We need to know what \( m \) was, to compute the result.
\[ f_1 \]

\[
\text{print (null (}\ f_1 \ m \text{))}
\]
print (null (f1 m))

needs the Bool to print it

(IO is the prime directive 🎉🎉🎉)
evaluate!

null \ (f1 \ m)
case f1 m of
  [] → True
  _:_ → False
Pattern matching drives evaluation
m is **NOT** evaluated, because we didn’t pattern match on it

```
case [m, m] of
  [] → True
  _:_ → False
```
in the end, m is never evaluated

```haskell
case m : (m : []) of
  []  -> True
_:_  -> False

\[\downarrow\]
False
```
evaluated enough to case
print (null (f2 m))
null (f2 m)
case \texttt{f2 m} of
  [] \rightarrow \text{True}
  _:_ \rightarrow \text{False}
\[ f_2 \]

\[
\text{case } \begin{cases}
\text{Nothing} & \rightarrow [\ ] \\
\text{Just } x & \rightarrow [x]
\end{cases}
\text{ of } m
\]

\[
[] \rightarrow \text{True} \\
\_ : \_ \rightarrow \text{False}
\]

\( m \) is evaluated!
The rules:

- Expressions are evaluated on pattern match...

- ... but only enough to make the match go through

- Initial evaluation is triggered on IO
- Built-ins pattern match too!
Recall: \( \text{any}::(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool} \)
\( \text{any} \ p = \text{or} \ . \ \text{map} \ p \)

\[
\text{print} (\text{any} \ (>1) \ [\emptyset..]))
\]
\text{infinite list!}
print (any (≥1) [∅..])
any

\[ \downarrow \]

\[ \text{or \ (map \ (\geq 1) \ [\emptyset ..])} \]

I expanded away the \( . \) as well
any

```haskell
  case (map (≥1) [ø..]) of
    [] → False
    True:_ → True
    False: rest → or rest
```
\[
\text{case } [\alpha:] \text{ of } \\
\text{[] } \rightarrow \text{ False} \quad \text{True: } - \rightarrow \text{ True} \\
\]

\[
\forall x \geq 2 \in \mathbb{Z} \rightarrow \text{map } (\geq 1) \text{ of } x :: \text{xs} \rightarrow (x :: \text{xs}) \text{ of } x :: \text{rest} \\
\]

\[
\text{we won't evaluate all of } [\alpha:], \text{ just the first element} \\
\]

\[
\text{hue} \\
\]
any

\[
\text{case } \emptyset : [1..] \text{ of } \\
[] \rightarrow [] \\
(x:xs) \rightarrow (x \geq 1): \text{ map } (\geq 1) \ xs
\]

[] \rightarrow \text{False}

True:_ \rightarrow \text{True}

False: \text{rest} \rightarrow \text{or rest}
any

```
\text{case} \quad (x \geq 1) : \text{map} (\geq 1)
```

```
\text{thunk}
```

```
\text{bound!}
```

```
\text{of}
```

```
\text{this match is refuted}
```

```
\text{True: } _ \rightarrow \text{True}
```

```
\text{False: rest } \rightarrow \text{or rest}
```

```
\text{[ ]} \rightarrow \text{False}
```
any

\[
\text{case } (\emptyset \geq 1): \text{ map } (\geq 1) \ [1..] \ \text{ of}
\]

\[
\text{True: } _- \rightarrow \text{ True}
\]

\[
\text{False: rest } \rightarrow \text{ or rest}
\]

must evaluate the head of the list
any

case False: map (≥1) [1..] of

True:_ → True
False: rest → or rest

must evaluate the head of the list
any

map (≥1) [1..]

bound

or rest

the tail is not evaluated yet!
any

\[ \varnothing \ldots \] drifts to 1, then to \[ [2\ldots] \].

\[ \text{map} \ (\geq 1) \] from True to \[ \text{map} \ (\geq 1) \] in the end.

\[ \text{map} \ (\geq 1) \] from False to True.

\text{or}
Stream combinators

\textbf{map} :: (a \to b) \to [a] \to [b]

\textbf{filter} :: (a \to \text{Bool}) \to [a] \to [a]

\textbf{++} :: [a] \to [a] \to [a]

\textbf{head} :: [a] \to a

\textbf{tail} :: [a] \to [a]

\textbf{and} :: [\text{Bool}] \to \text{Bool}

\textbf{zip} :: [a] \to [b] \to [(a, b)]

\textbf{foldr} :: (a \to b \to b) \to b \to [a] \to b
foldr
foldr

foldr f z [a, b]  ⇒  a

(f evaluate? b z

(evaluate to weak head normal form)
foldr

foldr \(f, z\) \([a, b]\) 

evaluate?

depends on \(f\)!
foldr as map

map g [a, b]  \Rightarrow  g a

f x xs = g x : xs
z = []

Neither evaluated!
foldr as filter

\[ \text{filter } p \ [a,b] \]

Q: Was \( a \) evaluated?

\[ f \ x \ xs = \text{if } p \ x \ \text{then } x : xs \ \text{else } xs \]

\[ z = [] \]
foldr as or

f x xs = if x then True else xs
z = False
foldr as \text{sum}???

\text{sum? } [a, b]

\begin{align*}
f(x \ x) &= x + x \ xs \\
&\vphantom{f(x \ x)} \\
&\vphantom{f(x \ x)} \\
&\vphantom{f(x \ x)} \\
&\vphantom{f(x \ x)} z = \emptyset
\end{align*}
foldr as sum ??

```plaintext
... e \ b d c a e + + +
...
```

this takes a lot of stack!

> foldr (+) Ø [1..1000000]

*** Exception: stack overflow
The Dark Side of Lazy Evaluation
Lazy friendly functions like foldr will **Stack overflow** if you try to evaluate them all at once.

\[ 1 + (2 + (3 + (4 + \ldots ))) \]

**must evaluate inside before outside**

Solution: Use an accumulator
$\text{foldl\, (+)\, } \emptyset\ [1..1000000]$
> foldl (+) ∅ [1..1000000]

*** Exception: stack overflow
foldl (+) ∅ [1...1000000]

foldl (+) (∅+1) [2...1000000]

foldl (+) ((∅+1)+2) [3...1000000]

\[\text{foldl doesn't evaluate the accumulator as you go}\]
Lazy evaluation may build up a large chain of deferred computation, leading to memory leak and stack overflow.

Solution: Evaluate as you go using strict functions (e.g. seq)
> foldl' (+) Ø [1..1000000]
500000500000

the tick is for strict
Bonus:

- How it's implemented

- Relation to generators (Python)
Representation of a thunk

<table>
<thead>
<tr>
<th>THUNK</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESERVED</td>
</tr>
<tr>
<td>PAYLOAD</td>
</tr>
</tbody>
</table>

Closure data needed to evaluate

Identifies this data as a thunk
(literally a function pointer to jump to)
Representation of a thunk

<table>
<thead>
<tr>
<th>THUNK</th>
<th>RESERVED</th>
<th>PAYLOAD</th>
</tr>
</thead>
</table>

evaluate
Representation of a thunk

<table>
<thead>
<tr>
<th>THUNK</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESERVED</td>
</tr>
<tr>
<td>PAYLOAD</td>
</tr>
</tbody>
</table>

Result
Representation of a thunk

atomically →

THUNK
PAYLOAD

Result
Representation of a thunk

atomically → INDIREDCTON

PAYLOAD dead

will be cleaned up by GC

no locks!
Notes

— This could race, resulting in duplicate work. Trick to solve this (at performance cost)

blackholing

— How to tell if data is thunk or not? Pointer tagging
Generators (in Python)

```python
def nats():
    i = 0
    while True:
        yield i
        i += 1
```
Generators (in Python)

def nats():
    i = 0
    while True:
        yield i
        i += 1

Suspended execution until next requested
Generators (in Python)

- Many Haskell idioms work
- But, generators are stateful!

```python
xs = range(0, 10)  # generator
for x in xs:
    print(x)
```

OK

```python
for x in xs:
    print(x)
```

Prints nothing