CSCI-UA.490

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)
```

evaluate me

What is strict evaluation?

```
let f x y = x+2 in

f 5 (29^35792)
```

$$\underset{\text{evaluate me}}{\underset{\uparrow}{(29\text{^}35792)}}$$

What is strict evaluation?

let f x y = x+2 in

wasted work!

$$\underbrace{f\ 5\ 140974576770288193}_{\text{evaluate f}}$$

What is lazy evaluation?

```
let f x y = x+2 in

f 5 (29^35792)
```

$\uparrow$

evaluate f

What is lazy evaluation?

```
let f x y = x+2 in
```

$$f\ \boxed{5}\ \boxed{(29\text{^}35792)}$$

$\uparrow$ $\uparrow$

suspended as <u>thunks</u>

What is lazy evaluation?

Don't try this with side effects!

- 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:

$$(\lambda x.\, x + x)\ (29\verb|^|35792)$$

$$\xrightarrow[\beta]{} 29\verb|^|35792 + 29\verb|^|35792$$

duplicate work!

Why does lazy evaluation matter?

Streams

Generators

$$o \rightarrow o \longrightarrow o \rightarrow o \rightarrow \cdots$$

Monotonicity

$$\boxed{\bot} \leq \boxed{1:\bot} \leq \boxed{1:2:\bot} \leq \cdots$$

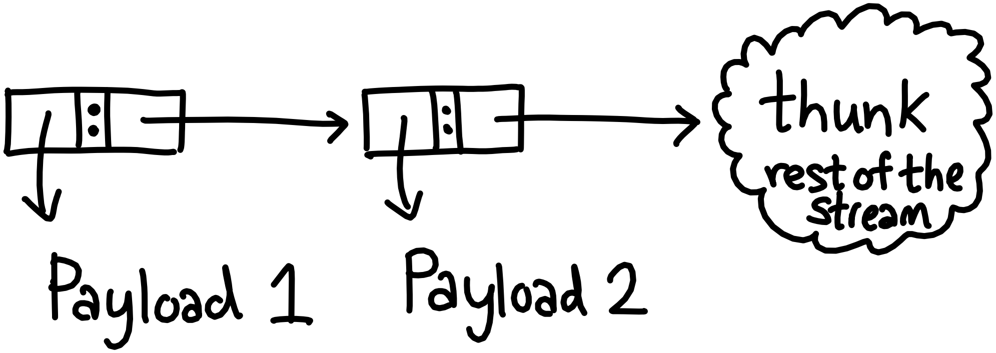
Functional programming

DSLs
Compositionality
Cyclic data structures

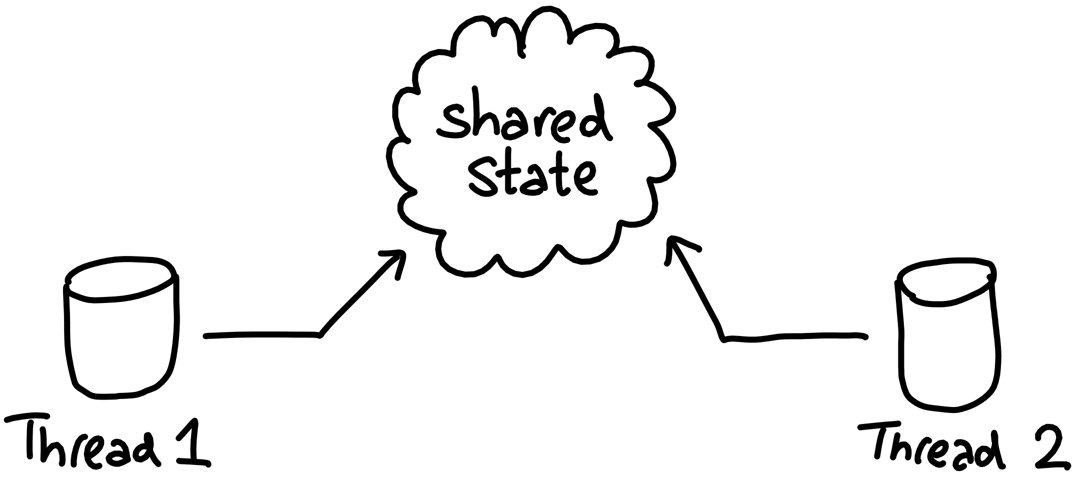
Why does lazy evaluation matter? Streams

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

- Streams are a lazy data structure



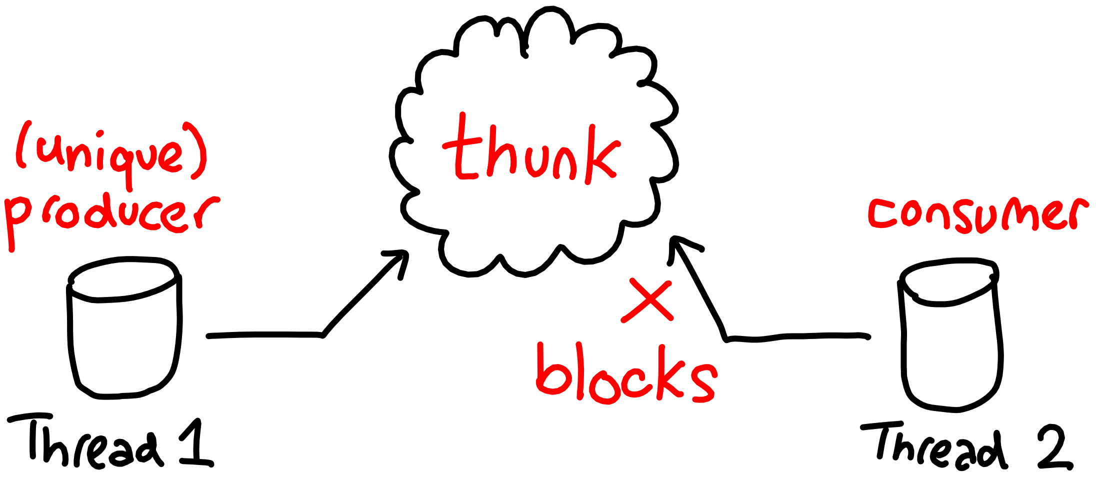
Why does lazy evaluation matter? Monotonicity



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 <u>impossible</u> to race with thunks
(Correct-by-construction abstraction)

(I said FP would be good for concurrency)

Why does lazy evaluation matter? FP

DSLs

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

vs.

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

DSL friendly languages have compact syntax for closures

Why does lazy evaluation matter? FP

Compositionality

```
any :: (a -> Bool) -> [a] -> Bool
any p = or . map p
```

$$\begin{tikzcd} \texttt{or} & \texttt{map p} & {[a] \to [Bool]} \arrow[l, bend right] \\ & {[Bool] \to Bool} \arrow[ul, bend left] & \end{tikzcd}$$

This is lazy! As soon as we find the first 'a' that satisfies the predicate, we stop

Why does lazy evaluation matter? FP

Compositionality

```
any :: (a → Bool) → [a] → Bool
any p = or . 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

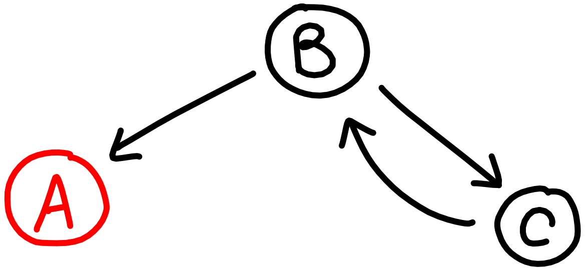
$$\begin{tikzcd}
 & B \arrow[dl] \arrow[dr] & \\
A & & C \arrow[ul, bend left]
\end{tikzcd}$$

Why does lazy evaluation matter? FP

Cyclic data structures

Graphs are difficult in a pure language

```
let a = Node []
```



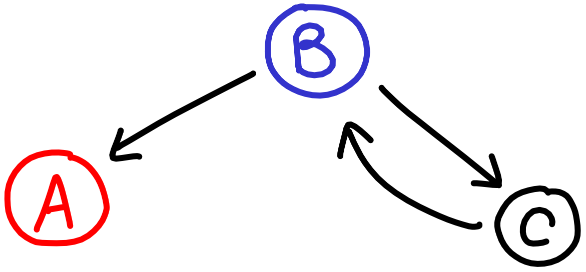
$$\begin{tikzcd}
 & B \arrow[dl] \arrow[dr, bend left] & \\
A & & C \arrow[ul, bend left]
\end{tikzcd}$$

Why does lazy evaluation matter? FP

Cyclic data structures

Graphs are difficult in a pure language

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



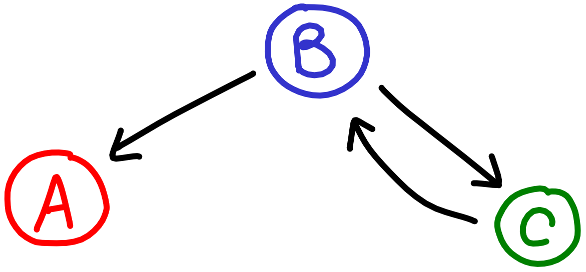
Why does lazy evaluation matter? FP

Cyclic data structures

Graphs are difficult in a pure language

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

normally, you'd use mutation to fill in edges after b is allocated

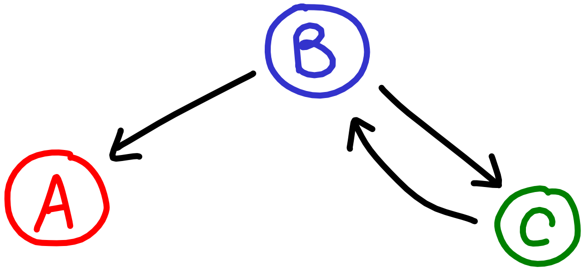


Why does lazy evaluation matter? FP

Cyclic data structures

Graphs are difficult in a pure language

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



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?

```
f1 :: Maybe a -> [Maybe a]
f1 m = [m, m]


f2 :: Maybe a -> [a]
f2 Nothing = []
f2 (Just x) = [x]
```

```
              f1
print (null (    m))
              f2
```

```
f1 :: Maybe a → [Maybe a]
f1 m = [m, m]
```

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

```
f2 :: Maybe a → [a]
f2 Nothing = []
f2 (Just x) = [x]
```

↑ We need to know what m was, to compute the result

$\boxed{f1}$

```
print (null (f1 m))
```

f1

```
print (null (f1 m))
```

needs the Bool to print it

(IO is the prime directive)

$\boxed{f1}$

evaluate!

$\big\downarrow$

null (f1 m)

$\boxed{f1}$

```
case f1 m of
    [] -> True
    _:_ -> False
```

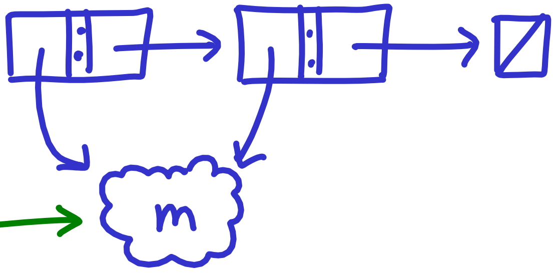
$\boxed{\text{f1}}$

evaluate!

```
case f1 m of
  [] -> True
  _:_ -> False
```

Pattern matching drives evaluation

$\boxed{f1}$



m is <u>NOT</u> evaluated, because we didn't pattern match on it

```
case [m, m] of
  [] -> True
  _:_ -> False
```

$\boxed{\text{f1}}$

in the end,
m is never
evaluated

evaluated enough to case

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

$\downarrow$

False

$\boxed{\text{f2}}$

```
print (null (f2 m))
```

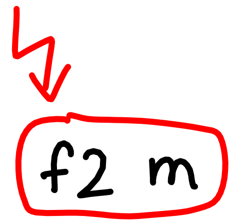
$\boxed{f2}$

$\downarrow$

null (f2 m)

$\boxed{\text{f2}}$

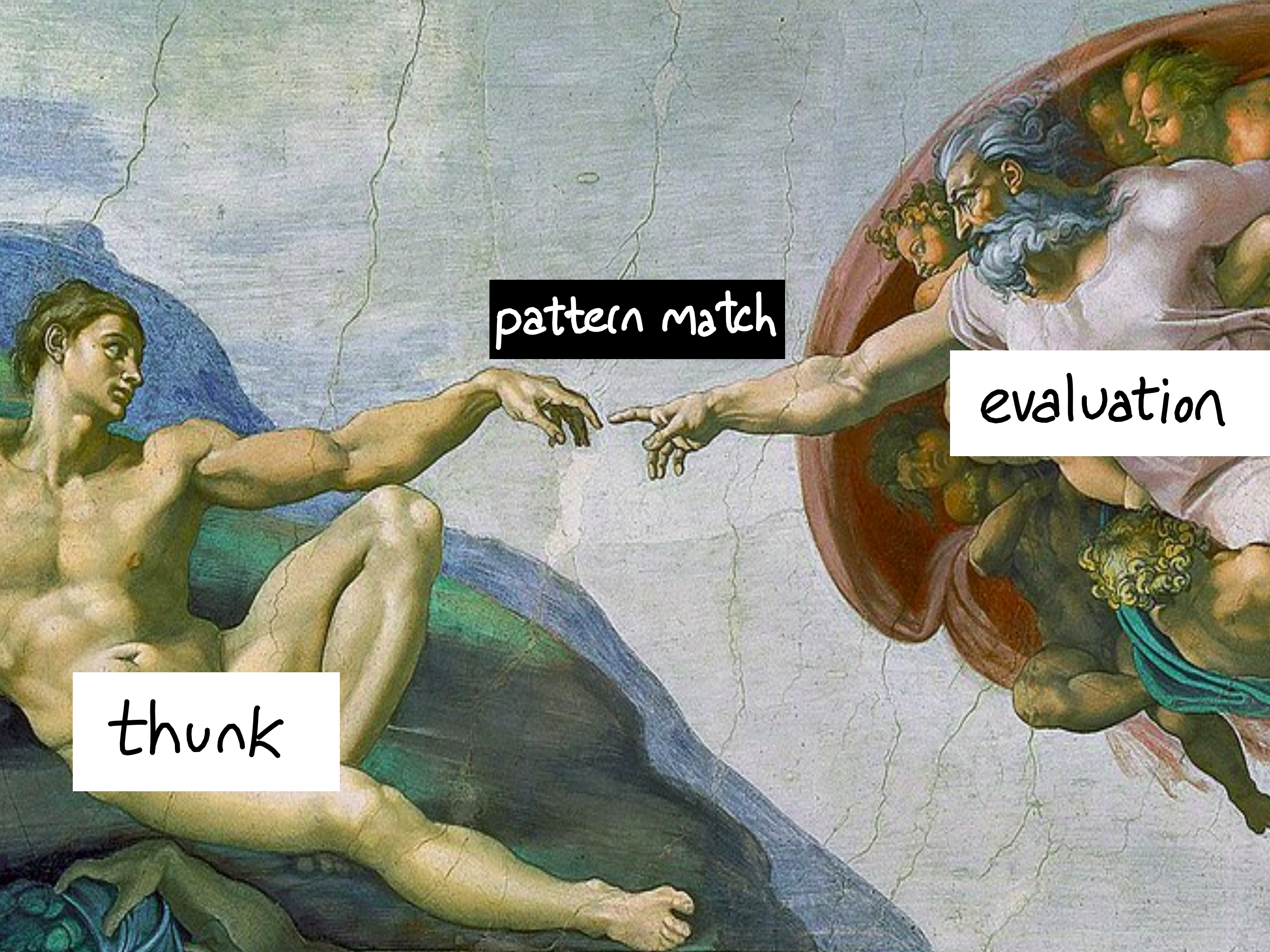
```
case f2 m of
    [] -> True
    _:_ -> False
```



$\boxed{\text{f2}}$

m is evaluated!

```
case (case m of
          Nothing -> []
          Just x -> [x]) of
    [] -> True
    _:_ -> False
```


pattern match
evaluation
thunk

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!

$\boxed{\text{any}}$

Recall:
```
any :: (a -> Bool) -> [a] -> Bool
any p = or . map p
```

$\texttt{print (any (>1) }\underbrace{\texttt{[0..]}}_{\text{infinite list!}}\texttt{)}$

any

```
print (any (≥1) [0..])
```

$\boxed{\text{any}}$

any (≥ 1) $[\emptyset..]$

$\boxed{\text{any}}$

```
or (map (≥1) [0..])
```

I expanded away the . as well

$\boxed{\text{any}}$

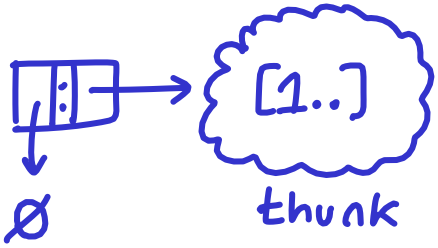
```
case (map (≥1) [0..]) of
  [] → False
  True:_ → True
  False:rest → or rest
```

$\boxed{\text{any}}$

we won't evaluate all of [0..], just the first element

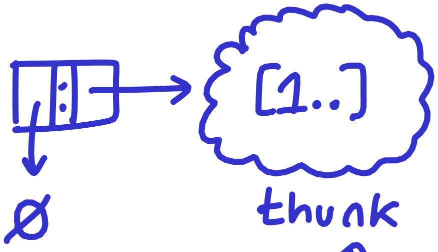
```
case ( case [0..] of
         []     -> []
         (x:xs) -> (x ≥ 1) : map (≥1) xs ) of
  []         -> False
  True:_     -> True
  False:rest -> or rest
```

any



```
case ( case ∅ : [1..] of
           [] -> []
           (x:xs) -> (x ≥ 1) : map (≥1) xs
     ) of
    [] -> False
    True:_ -> True
    False:rest -> or rest
```

any



bound!

```
case (x ≥ 1) : map (≥1) xs of

  ~~[] → False~~
  True : _ → True
  False : rest → or rest
```

this match is refuted

any

```
case (Ø≥1) : map (≥1) [1..] of

    True : _     → True
    False : rest → or rest
```

must evaluate the head of the list

$\boxed{\text{any}}$

```
case  False : map (≥1) [1..]    of

   ~~True : _  → True~~
   False : rest → or rest
```

must evaluate the head of the list

any

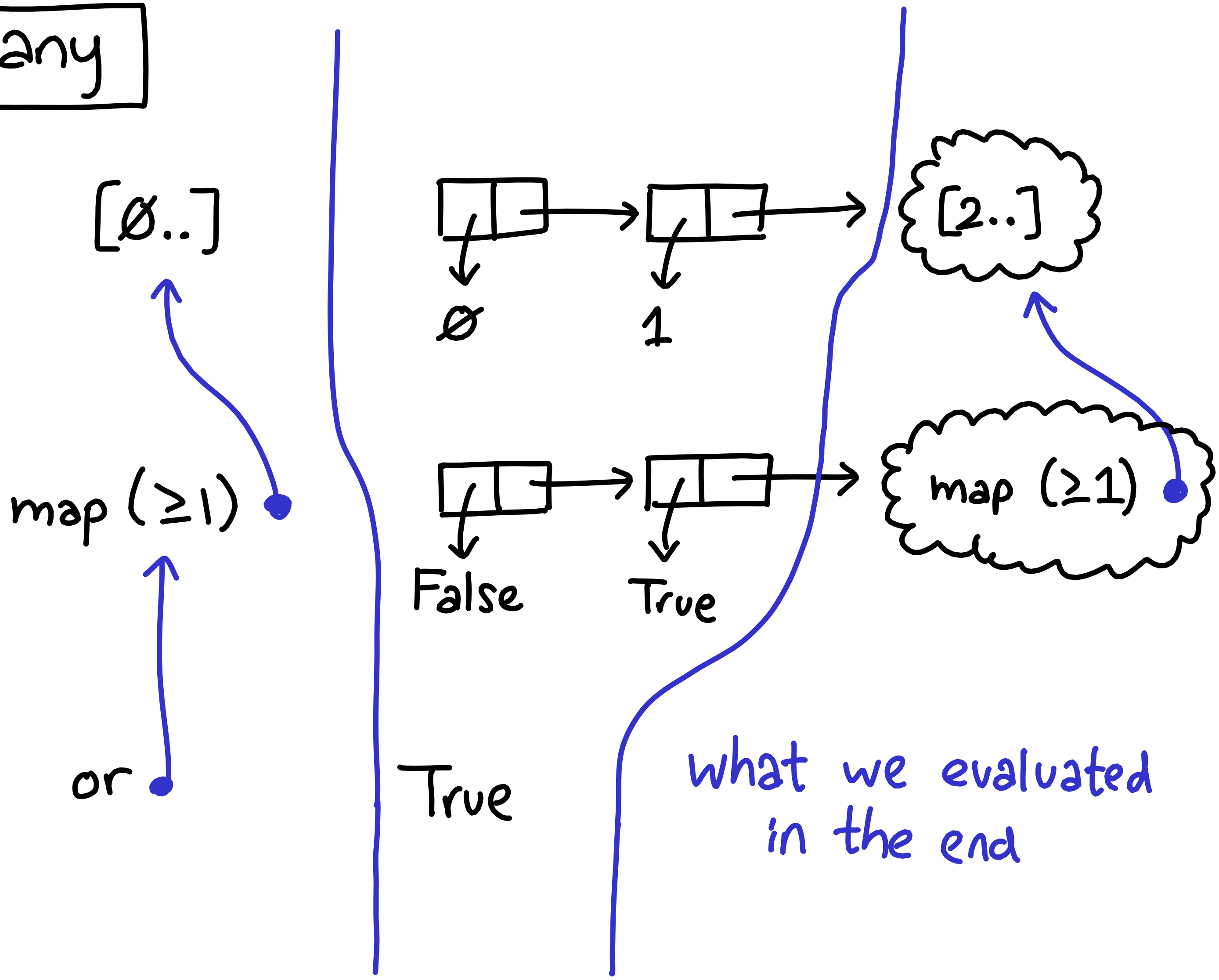
map (≥1) [1..]

the tail is not evaluated yet!

bound

or rest

$\boxed{\text{any}}$



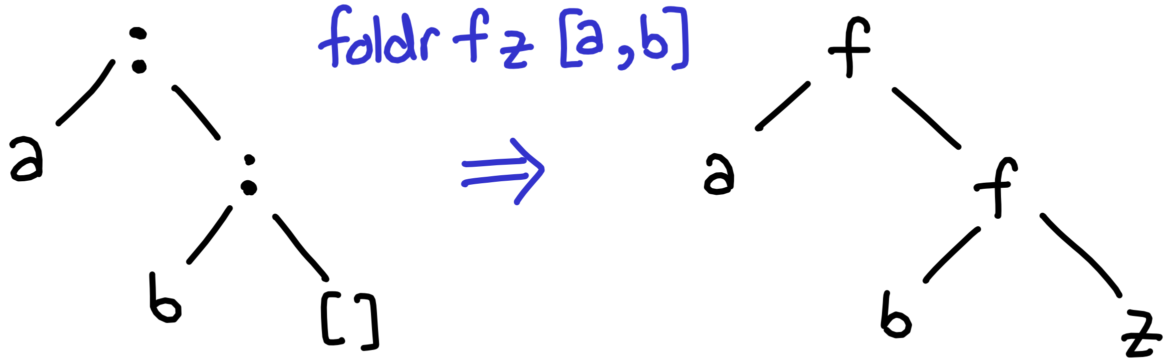
Stream combinators

All lazy

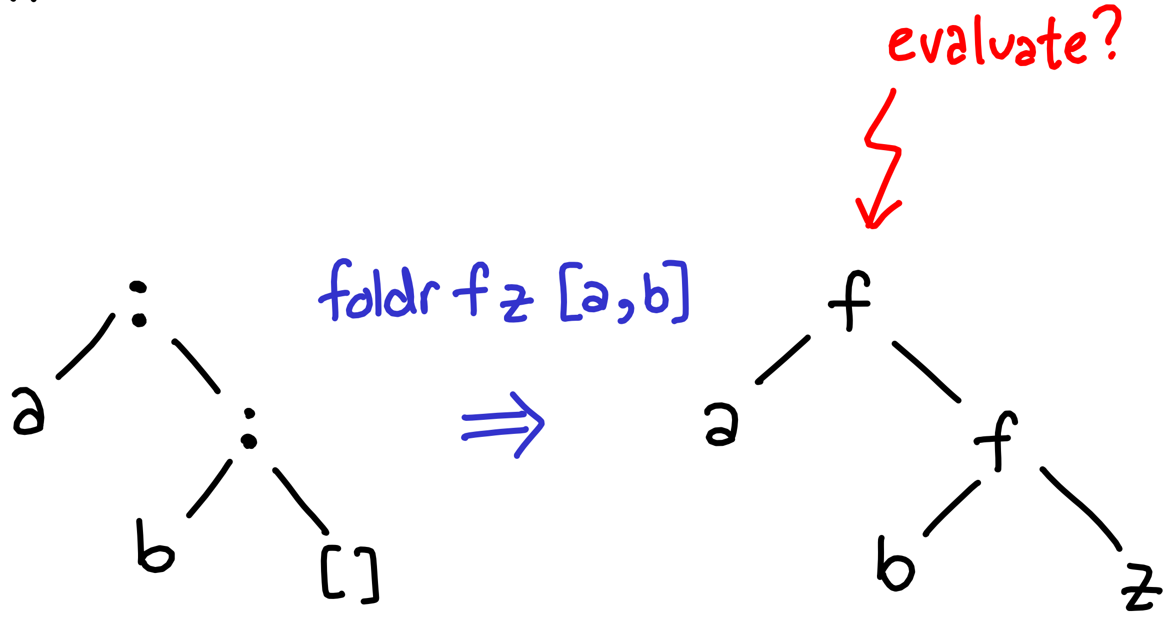
Falls out naturally from singly linked lists

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
head :: [a] -> a
tail :: [a] -> [a]
and :: [Bool] -> Bool
zip :: [a] -> [b] -> [(a,b)]
foldr :: (a -> b -> b) -> b -> [a] -> b
```

foldr

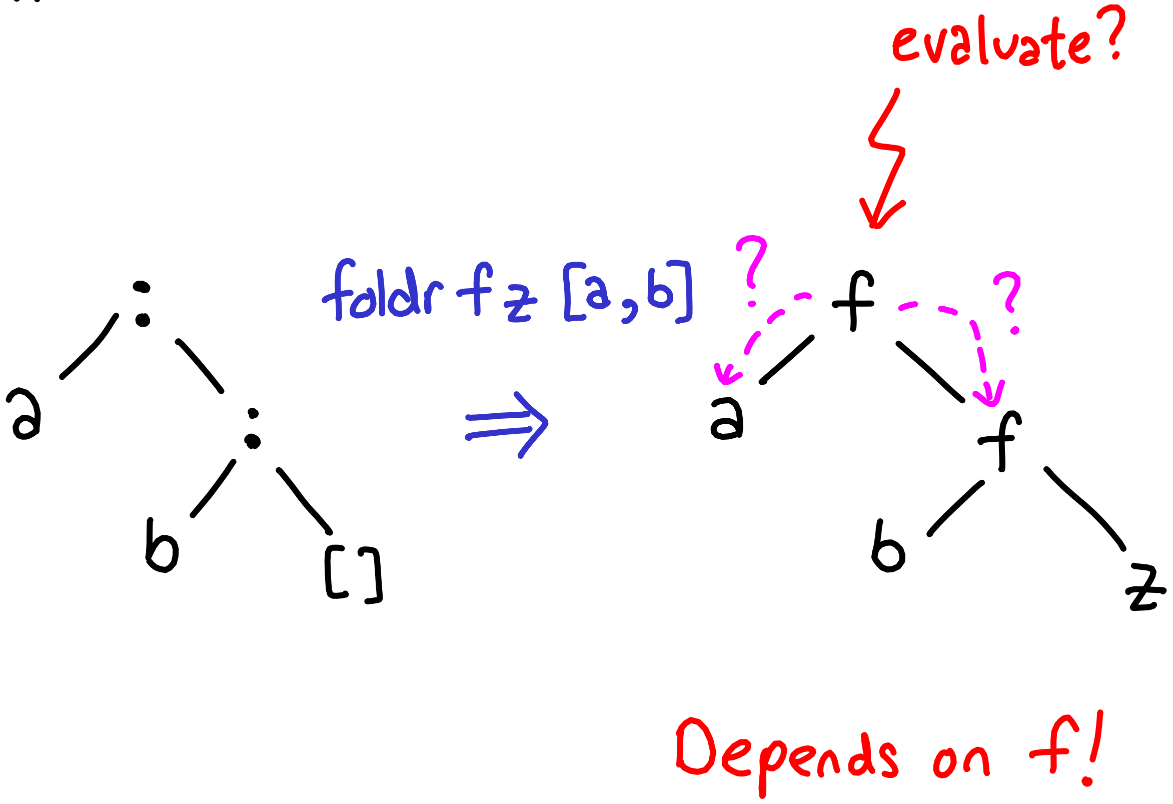


foldr

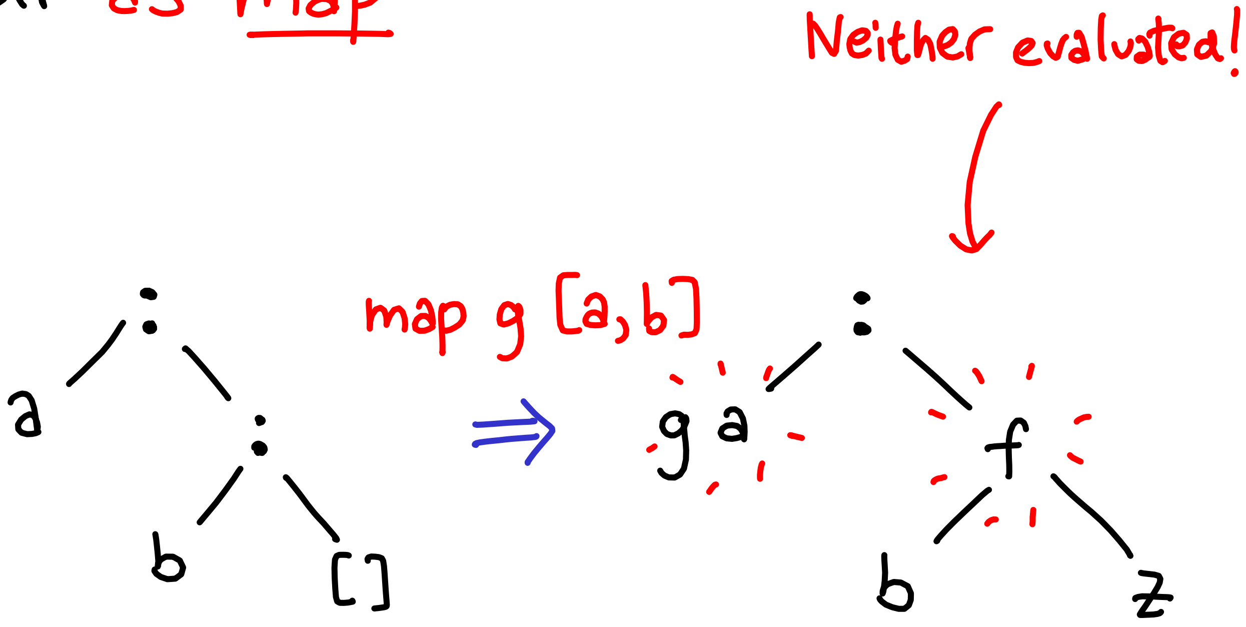


(evaluate to weak head normal form)

foldr

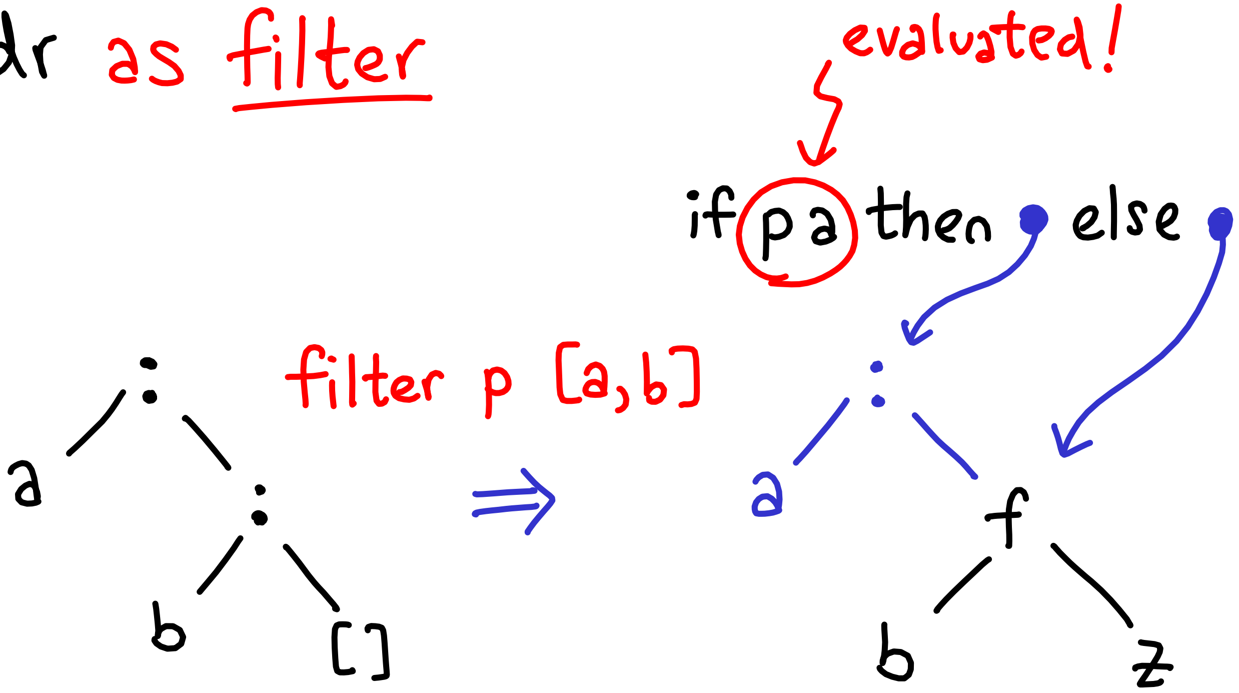


foldr as map



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

foldr as filter

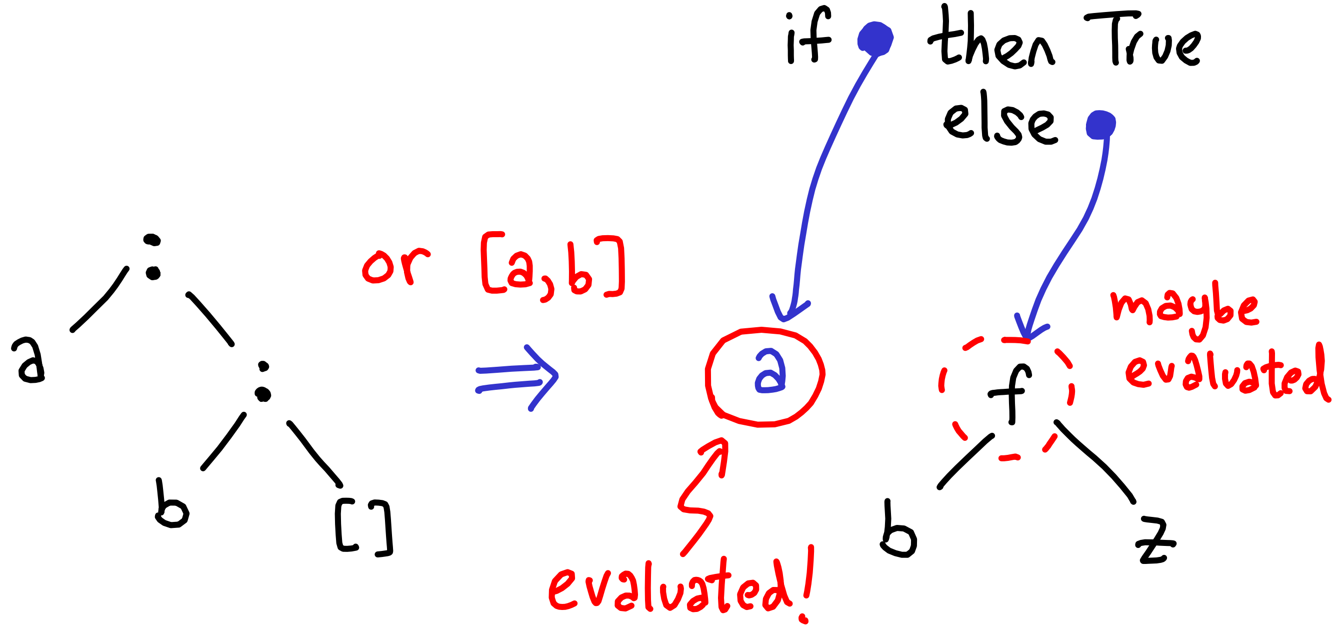


Q: Was a evaluated?

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

$z = []$

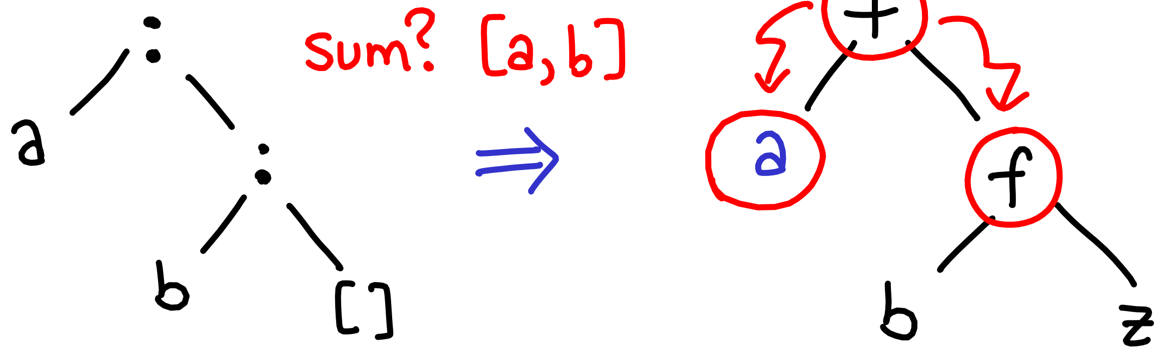
foldr as or



f x xs = if x then True else xs

z = False

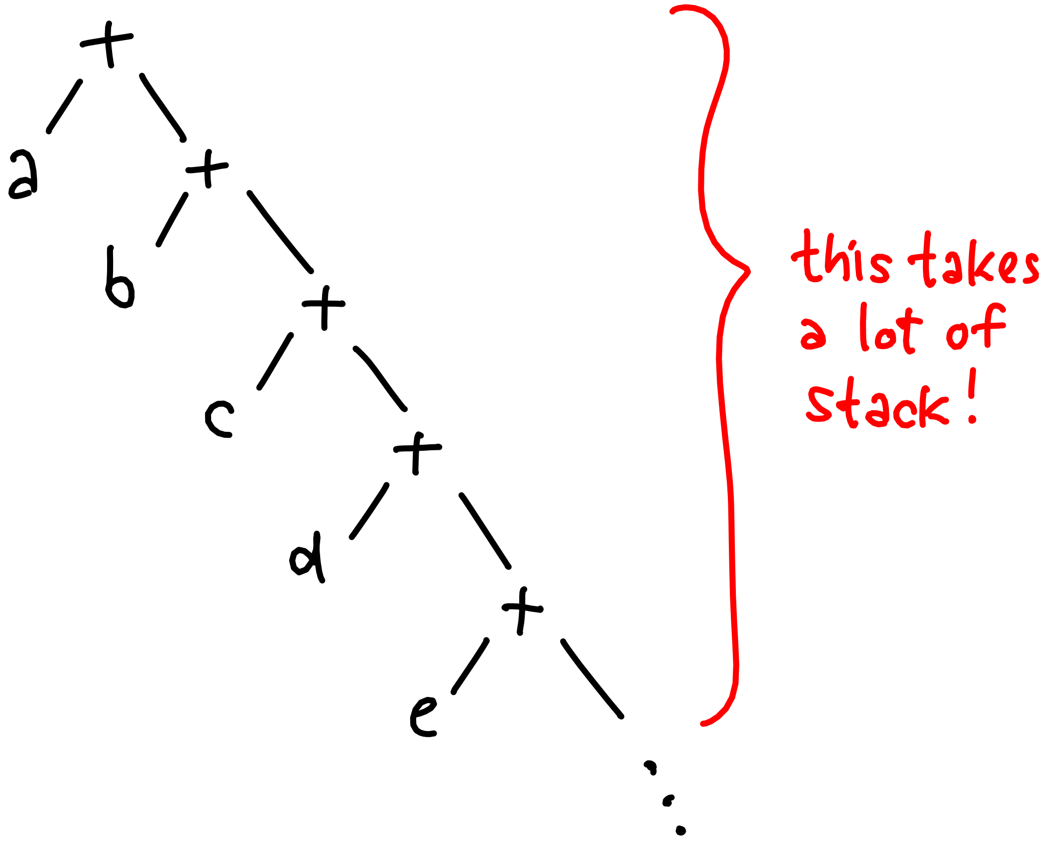
foldr as sum???



$f\ x\ xs = x + xs$

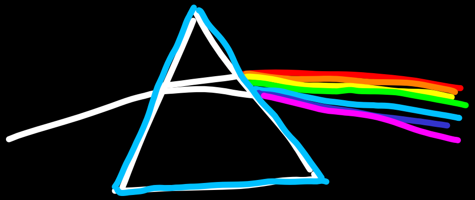
$z = \emptyset$

foldr as sum???



```
> foldr (+) 0 [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.

$$1 + (2 + (3 + \underbrace{(4 + \ldots}_{\text{must evaluate inside before outside}})))$$

Solution: Use an accumulator

```
> foldl (+) 0 [1..1000000]
```

$$\underbrace{((0+1)+2)}_{\text{accumulator}}+3)+\ldots$$

```
> foldl (+) 0 [1..1000000]
*** Exception: stack overflow
```

```
foldl (+) 0 [1...1000000]
```

not used!

↓

```
foldl (+) (0+1) [2...1000000]
```

↓

```
foldl (+) ((0+1)+2) [3...1000000]
```

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)

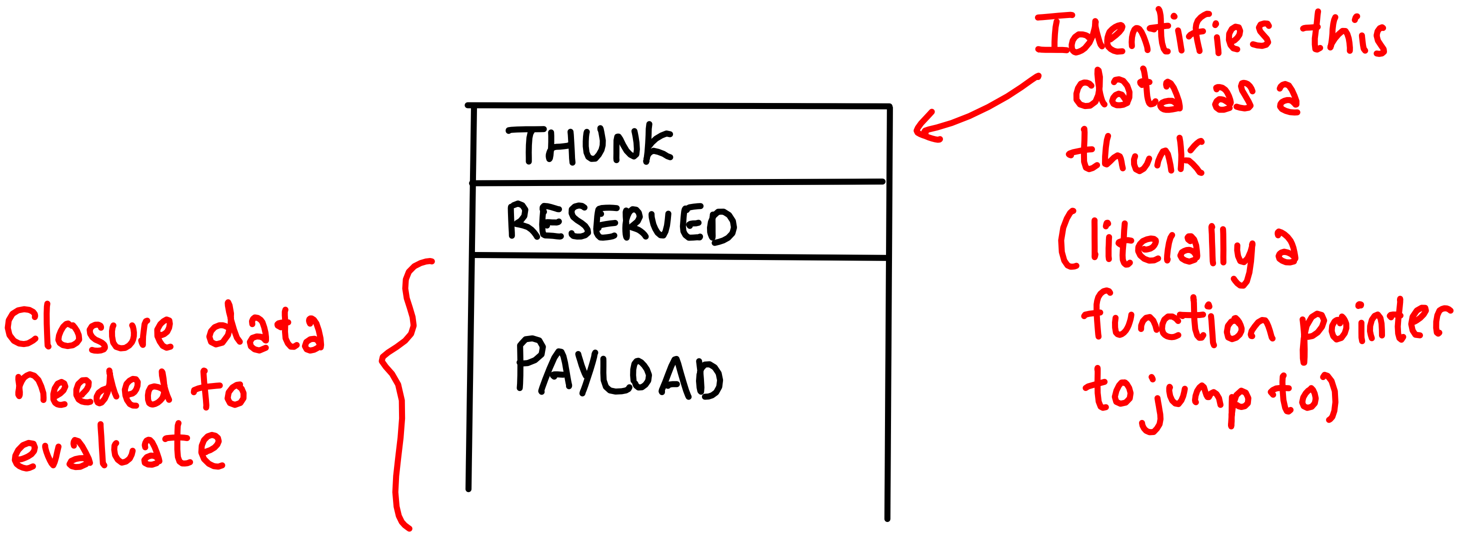
the tick is for strict

```
> foldl' (+) 0 [1..1000000]
500000500000
```

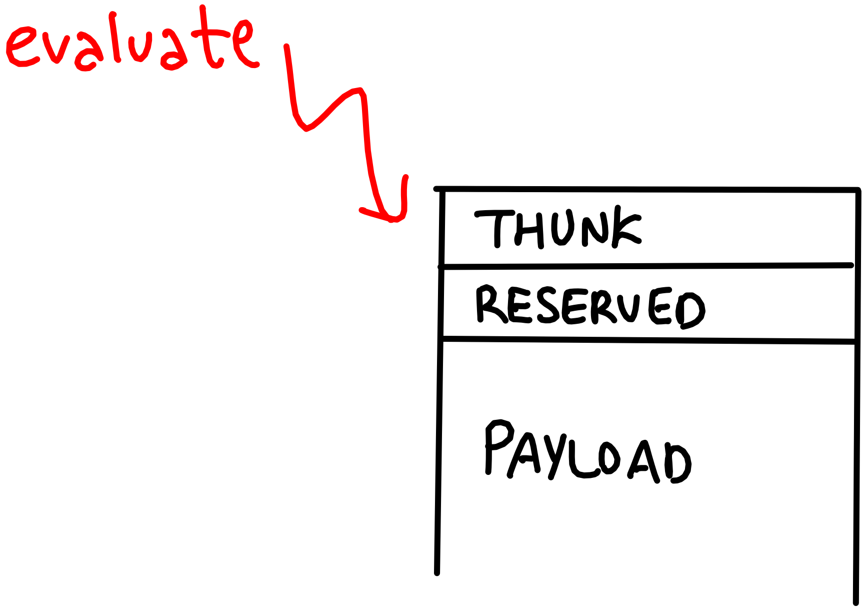
Bonus:

- How it's implemented
- Relation to generators (Python)

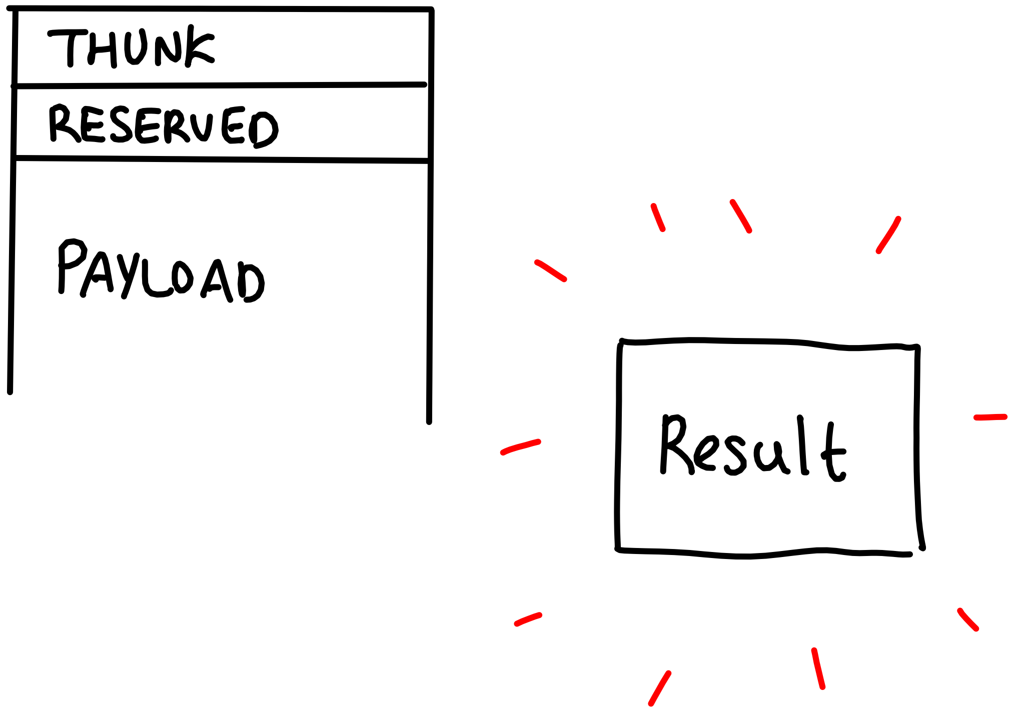
Representation of a thunk


THUNK
RESERVED
PAYLOAD
Identifies this data as a thunk
(literally a function pointer to jump to)
Closure data needed to evaluate

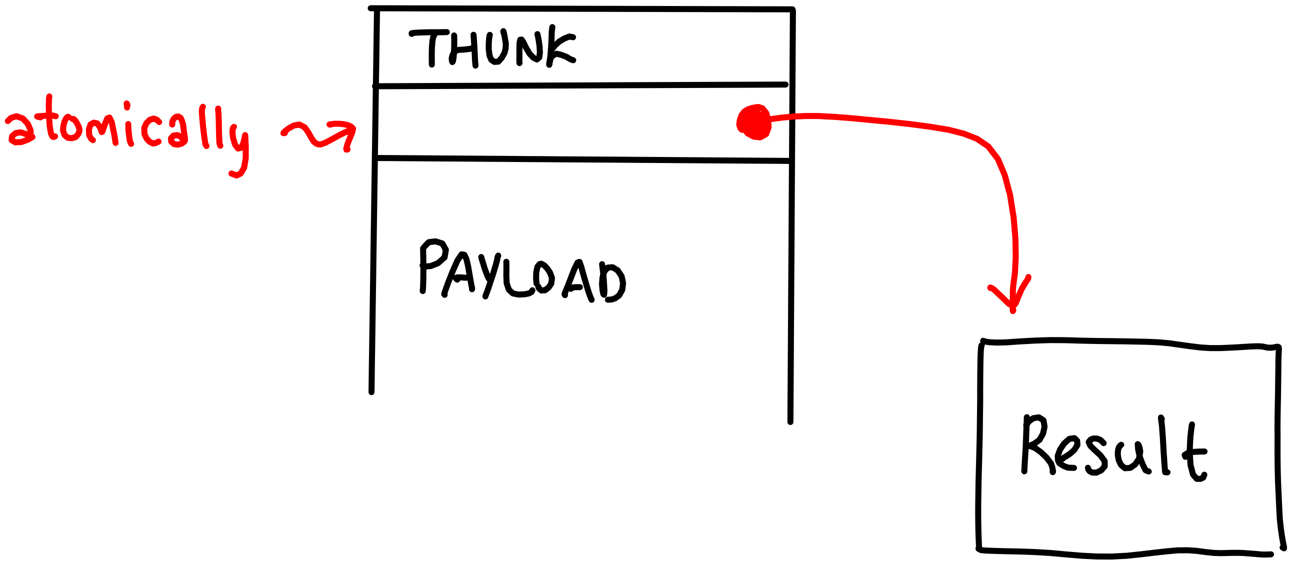
Representation of a thunk



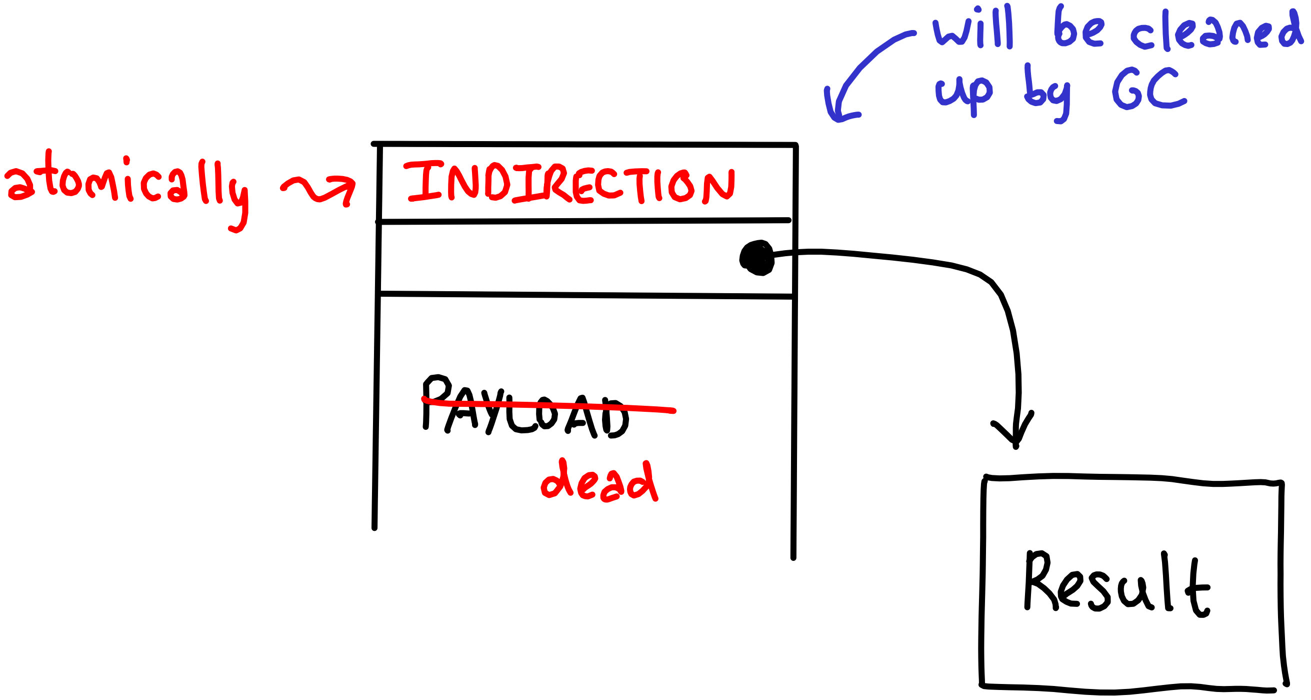
Representation of a thunk



Representation of a thunk



Representation of a thunk



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)

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

Generators (in Python)

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

← suspends execution until next requested

Generators (in Python)

- Many Haskell idioms work

- But, generators are stateful!

```
xs = range(Ø, 10)      ← generator
for x in xs:
    print(x)           OK ✓
for x in xs:
    print(x)           Prints nothing
```