Lazy Evaluation

ZuriHac 2023

Andres Löh

2023-06-11 —

Well-Typed
The Haskell Consultants

About Well-Typed

- Well-Typed is a Haskell consultancy company, established in 2008
- Team of about 20 Haskell experts
- Wide variety of clients

- GHC and tooling maintenance, development and support
- Haskell software development and consulting
- On-site and remote training courses

GHC support and maintenance

`https://well-typed.com/blog/2022/11/funding-ghc-maintenance/`

About me

- Using Haskell since about 1997
- Studied mathematics in Konstanz, PhD in Computer Science at Utrecht 2004
- At Well-Typed since 2010
- Living in Regensburg, Germany

Haskell Interlude

https://haskell.foundation/podcast/

Well-Typed

Haskell Unfolder

https://www.youtube.com/@well-typed

Next episode on Wednesday, 14 June, on a topic related to this talk!

Repository

This presentation and the code samples are available from

https://github.com/well-typed/lazy-evaluation-zurihac-2023

The plan

- Look at lazy evaluation and try to reason about simple programs.
- Build an intuition for lazy evaluation.
- Discuss some common pitfalls.

The plan

- Look at lazy evaluation and try to reason about simple programs.
- Build an intuition for lazy evaluation.
- Discuss some common pitfalls.

Not:

- Complete in any sense.
- Dive deep into GHC-specific optimisations.
- Learn how to track down space leaks in large code bases.

Informal introduction

Lazy evaluation

What is lazy evaluation?

Lazy evaluation

What is lazy evaluation?

- evaluate as little as possible, just when needed, and …

Lazy evaluation

What is lazy evaluation?

- evaluate as little as possible, just when needed, and ...
- share computation results if they are needed multiple times.

Lazy evaluation

What is lazy evaluation?

- evaluate as little as possible, just when needed, and ...
- share computation results if they are needed multiple times.

What is a space leak?

Lazy evaluation

What is lazy evaluation?

- evaluate as little as possible, just when needed, and ...
- share computation results if they are needed multiple times.

What is a space leak?

A situation where memory is retained by the program unexpectedly long.

Lazy evaluation

Why do we evaluate anything at all?

Lazy evaluation

Why do we evaluate anything at all?

- Some result we are interested in creates demand on other results.
- Demand is propagated through functions and language constructs such as `case` (or more generally pattern matching).

Well-Typed

Lazy evaluation

Why do we evaluate anything at all?

- Some result we are interested in creates demand on other results.
- Demand is propagated through functions and language constructs such as `case` (or more generally pattern matching).

We will try to make these points more precise throughout the lecture.

Example 1: null

A first example

```haskell
example1 :: Int -> Bool
example1 n = null [0 .. n]
```

How much space does this use (in terms of `n`)?

Looking at definitions

Let’s start with our own definitions.

```
null :: [a] -> Bool
null []      = True
null (_ : _) = False
```

Well-Typed

Looking at definitions

Let's start with our own definitions.

```
null :: [a] -> Bool
null []      = True
null (_ : _) = False
```

```
enumFromTo :: Int -> Int -> [Int]
enumFromTo l u =
  if l > u
    then []
    else l : enumFromTo (l + 1) u
```

In Haskell, `[m .. n]` is syntactic sugar for `enumFromTo m n` .

Equational reasoning

Let’s assume `n = 2` :

```
    null (enumFromTo 0 2)
```

Equational reasoning

Let’s assume `n = 2` :

```
    null (enumFromTo 0 2)
```

Well-Typed

Equational reasoning

Let's assume `n = 2` :

```
  null (enumFromTo 0 2)
= null (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2)
```

Well-Typed

Equational reasoning

Let's assume `n = 2` :

```
  null (enumFromTo 0 2)
= null (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2)
```

Equational reasoning

Let's assume `n = 2` :

```
   null (enumFromTo 0 2)
 = null (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2)
 = null (if False then [] else 0 : enumFromTo (0 + 1) 2)
```

Equational reasoning

Let's assume `n = 2` :

```
   null (enumFromTo 0 2)
 = null (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2)
 = null (if False then [] else 0 : enumFromTo (0 + 1) 2)
```

Equational reasoning

Let's assume `n = 2` :

```
   null (enumFromTo 0 2)
 = null (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2)
 = null (if False then [] else 0 : enumFromTo (0 + 1) 2)
 = null (0 : enumFromTo (0 + 1) 2)
```

Equational reasoning

Let's assume `n = 2` :

```
  null (enumFromTo 0 2)
= null (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2)
= null (if False then [] else 0 : enumFromTo (0 + 1) 2)
= null (0 : enumFromTo (0 + 1) 2)
```

Well-Typed

Equational reasoning

Let's assume `n = 2` :

```
  null (enumFromTo 0 2)
= null (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2)
= null (if False then [] else 0 : enumFromTo (0 + 1) 2)
= null (0 : enumFromTo (0 + 1) 2)
= False
```

Equational reasoning

Let's assume `n = 2` :

```
  null (enumFromTo 0 2)
= null (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2)
= null (if False then [] else 0 : enumFromTo (0 + 1) 2)
= null (0 : enumFromTo (0 + 1) 2)
= False
```

Reduction sequence does not depend on `n` ,
only on `0 > n` being `False` .

Equational reasoning

Let's assume `n = 2` :

```
   null (enumFromTo 0 2)
 = null (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2)
 = null (if False then [] else 0 : enumFromTo (0 + 1) 2)
 = null (0 : enumFromTo (0 + 1) 2)
 = False
```

Reduction sequence does not depend on `n` ,
only on `0 > n` being `False` .

Answer to our original question is **constant space** (and time).

Well-Typed

Redexes

```
null (0 : enumFromTo (0 + 1) 2)
```

Redexes

```
null (0 : enumFromTo (0 + 1) 2)
```

Well-Typed

Redexes

```
null (0 : enumFromTo (0 + 1) 2)
```

Well-Typed

Redexes

```
null (0 : enumFromTo (0 + 1) 2)
```

Well-Typed

Redexes

```
null (0 : enumFromTo (0 + 1) 2)
```

We generally have more than one **redex** (reducible expression).

One aspect of lazy evaluation is that we are generally choosing the **outermost** redex.

Lightweight measuring

- Write the program.
- Run with different inputs (for `n`) and observe memory consumption.
- Use GHC RTS flags to get helpful info about memory use.

Demand?

Why does anything happen at all?

Demand?

Why does anything happen at all?

- We want to **print** the resulting `Bool` .

Demand?

Why does anything happen at all?

- We want to **print** the resulting `Bool` .
- In order to print it, we have to know it.

Demand?

Why does anything happen at all?

- We want to **print** the resulting `Bool` .
- In order to print it, we have to know it.
- So we have to evaluate the call to `null` .

Demand?

Why does anything happen at all?

- We want to **print** the resulting `Bool` .
- In order to print it, we have to know it.
- So we have to evaluate the call to `null` .
- Why can't we reduce `null (enumFromTo 0 2)` directly?

Pattern matching

```
null :: [a] -> Bool
null []      = True
null (_ : _) = False
```

The pattern match on the input drives evaluation, i.e., it **propagates demand**.

Just enough evaluation

As can be observed by the reduction

```
   null (0 : enumFromTo (0 + 1) 2)
 = False
```

revealing the top-level constructor is sufficient to reduce `null` .

Just enough evaluation

As can be observed by the reduction

```
   null (0 : enumFromTo (0 + 1) 2)
 = False
```

revealing the top-level constructor is sufficient to reduce `null` .

An expression is in **weak head-normal form (WHNF)** if it is a constructor application (or a lambda).

Just enough evaluation

As can be observed by the reduction

```
   null (0 : enumFromTo (0 + 1) 2)
 = False
```

revealing the top-level constructor is sufficient to reduce `null`.

An expression is in **weak head-normal form (WHNF)** if it is a constructor application (or a lambda).

Intuitively, if any evaluation is needed at all, then evaluating up to weak head-normal form is the least amount of evaluation that can enable new reduction opportunities.

How much evaluation?

So what about each of the following?

```
null (repeat 1)
null undefined
null (1 : undefined)
null (undefined : undefined)
null (let x = x in x)
```

Well-Typed

Aside: strict functions

A function `f` is called **strict** if and only if `f ⊥ = ⊥`.

(Here, `⊥` is a special value that subsumes anything that crashes or loops, e.g. `undefined`.)

Aside: strict functions

A function `f` is called **strict** if and only if `f ⊥ = ⊥`.

(Here, `⊥` is a special value that subsumes anything that crashes or loops, e.g. `undefined`.)

Good:

Strictness is defined in terms of a function's **behaviour**, not its implementation.

Aside: strict functions

A function `f` is called **strict** if and only if `f ⊥ = ⊥`.

(Here, `⊥` is a special value that subsumes anything that crashes or loops, e.g. `undefined`.)

Good:

Strictness is defined in terms of a function's **behaviour**, not its implementation.

Not so good:

Some implications of the definition might be unintuitive.

The notion is not very precise, because there are "various degrees of strictness".

Well-Typed

Examples

Is `null` strict?

Examples

Is `null` strict?

Yes!

```
GHCi> null undefined
*** Exception: Prelude.undefined
```

Well-Typed

Examples

What is an example of a non-strict function?

Examples

What is an example of a non-strict function?

```
constZero :: a -> Int
constZero _ = 0
```

Examples

What is an example of a non-strict function?

```
constZero :: a -> Int
constZero _ = 0
```

```
GHCi> constZero undefined
0
```

Identity

```haskell
id :: a -> a
id x = x
```

Is `id` strict?

Identity

```
id :: a -> a
id x = x
```

Is `id` strict?

Yes!

```
GHCi> id undefined
*** Exception: Prelude.undefined
```

Identity

```haskell
id :: a -> a
id x = x
```

Is `id` strict?

Yes!

```
GHCi> id undefined
*** Exception: Prelude.undefined
```

Note that `id` propagates demand on the result to demand on its argument.

Another corner case

```
constError :: a -> b
constError _ = undefined
```

This function is **also strict**.

Example 2: null via equality

Changed definition of `null`

```
nullViaEq xs = xs == []

example2 :: Int -> Bool
example2 n = nullViaEq [0 .. n]
```

Does this change anything?

Definition of equality on lists

```
instance Eq a => Eq [a] where
  []       == []       = True
  (x : xs) == (y : ys) = x == y && xs == ys
  _xs      == _ys      = False
```

Equational reasoning

```
    nullViaEq (enumFromTo 0 2)
```

Well-Typed

Equational reasoning

```
nullViaEq (enumFromTo 0 2)
```

Well-Typed

Equational reasoning

```
   nullViaEq (enumFromTo 0 2)
 = enumFromTo 0 2 == []
```

Well-Typed

Equational reasoning

```
   nullViaEq (enumFromTo 0 2)
=  enumFromTo 0 2 == []
```

Equational reasoning

```
   nullViaEq (enumFromTo 0 2)
 = enumFromTo 0 2 == []
 = (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2) == []
```

Equational reasoning

```
   nullViaEq (enumFromTo 0 2)
 = enumFromTo 0 2 == []
 = (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2) == []
```

Well-Typed

Equational reasoning

```
  nullViaEq (enumFromTo 0 2)
= enumFromTo 0 2 == []
= (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2) == []
= (if False then [] else 0 : enumFromTo (0 + 1) 2) == []
```

Equational reasoning

```
  nullViaEq (enumFromTo 0 2)
= enumFromTo 0 2 == []
= (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2) == []
= (if False then [] else 0 : enumFromTo (0 + 1) 2) == []
```

Well-Typed

Equational reasoning

```
  nullViaEq (enumFromTo 0 2)
= enumFromTo 0 2 == []
= (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2) == []
= (if False then [] else 0 : enumFromTo (0 + 1) 2) == []
= (0 : enumFromTo (0 + 1) 2) == []
```

Equational reasoning

```
   nullViaEq (enumFromTo 0 2)
 = enumFromTo 0 2 == []
 = (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2) == []
 = (if False then [] else 0 : enumFromTo (0 + 1) 2) == []
 = (0 : enumFromTo (0 + 1) 2) == []
```

Well-Typed

Equational reasoning

```
   nullViaEq (enumFromTo 0 2)
 = enumFromTo 0 2 == []
 = (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2) == []
 = (if False then [] else 0 : enumFromTo (0 + 1) 2) == []
 = (0 : enumFromTo (0 + 1) 2) == []
 = False
```

Equational reasoning

```
   nullViaEq (enumFromTo 0 2)
 = enumFromTo 0 2 == []
 = (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2) == []
 = (if False then [] else 0 : enumFromTo (0 + 1) 2) == []
 = (0 : enumFromTo (0 + 1) 2) == []
 = False
```

Reduction steps change, but still independent of `n` .

Still **constant space** (and time).

Aside: which definition is better?

Which of the two definitions of `null` is better?

Aside: which definition is better?

Which of the two definitions of `null` is better?

The function `nullViaEq` has an unnecessarily restrictive type:

```
nullViaEq :: Eq a => [a] -> Bool
```

Example 3: self equality

Comparing a list to itself

```haskell
selfEqual :: Eq a => a -> Bool
selfEqual x = x == x
```

```haskell
example3 :: Int -> Bool
example3 n = selfEqual [0 .. n]
```

We are once again interested in the space behaviour.

Equational reasoning

This is where sharing comes into play:

```
    selfEqual (enumFromTo 0 2)
```

Equational reasoning

This is where sharing comes into play:

```
selfEqual (enumFromTo 0 2)
```

Well-Typed

Equational reasoning

This is where sharing comes into play:

```
   selfEqual (enumFromTo 0 2)
 = let x = enumFromTo 0 2 in x == x
```

Equational reasoning

This is where sharing comes into play:

```
   selfEqual (enumFromTo 0 2)
= let x = enumFromTo 0 2 in x == x
```

Equational reasoning

This is where sharing comes into play:

```
  selfEqual (enumFromTo 0 2)
= let x = enumFromTo 0 2 in x == x
= let x = 0 : enumFromTo (0 + 1) 2 in x == x
```

Equational reasoning

This is where sharing comes into play:

```
   selfEqual (enumFromTo 0 2)
 = let x = enumFromTo 0 2 in x == x
 = let x = 0 : enumFromTo (0 + 1) 2 in x == x
 = let x = 0 : x'; x' = enumFromTo (0 + 1) 2 in x == x
```

Equational reasoning

This is where sharing comes into play:

```
  selfEqual (enumFromTo 0 2)
= let x = enumFromTo 0 2 in x == x
= let x = 0 : enumFromTo (0 + 1) 2 in x == x
= let x = 0 : x'; x' = enumFromTo (0 + 1) 2 in x == x
```

Equational reasoning

This is where sharing comes into play:

```
   selfEqual (enumFromTo 0 2)
 = let x = enumFromTo 0 2 in x == x
 = let x = 0 : enumFromTo (0 + 1) 2 in x == x
 = let x = 0 : x'; x' = enumFromTo (0 + 1) 2 in x == x
 = let x = 0 : x'; x' = enumFromTo (0 + 1) 2
   in 0 == 0 && x' == x'
```

Equational reasoning

This is where sharing comes into play:

```
   selfEqual (enumFromTo 0 2)
 = let x = enumFromTo 0 2 in x == x
 = let x = 0 : enumFromTo (0 + 1) 2 in x == x
 = let x = 0 : x'; x' = enumFromTo (0 + 1) 2 in x == x
 = let x = 0 : x'; x' = enumFromTo (0 + 1) 2
   in 0 == 0 && x' == x'
 = let x' = enumFromTo (0 + 1) 2 in 0 == 0 && x' == x'
```

Equational reasoning

This is where sharing comes into play:

```
   selfEqual (enumFromTo 0 2)
 = let x = enumFromTo 0 2 in x == x
 = let x = 0 : enumFromTo (0 + 1) 2 in x == x
 = let x = 0 : x'; x' = enumFromTo (0 + 1) 2 in x == x
 = let x = 0 : x'; x' = enumFromTo (0 + 1) 2
   in 0 == 0 && x' == x'
 = let x' = enumFromTo (0 + 1) 2 in 0 == 0 && x' == x'
```

Equational reasoning

This is where sharing comes into play:

```
   selfEqual (enumFromTo 0 2)
 = let x = enumFromTo 0 2 in x == x
 = let x = 0 : enumFromTo (0 + 1) 2 in x == x
 = let x = 0 : x'; x' = enumFromTo (0 + 1) 2 in x == x
 = let x = 0 : x'; x' = enumFromTo (0 + 1) 2
   in 0 == 0 && x' == x'
 = let x' = enumFromTo (0 + 1) 2 in 0 == 0 && x' == x'
 = let x' = enumFromTo (0 + 1) 2 in True && x' == x'
```

Well-Typed

Equational reasoning

This is where sharing comes into play:

```
  selfEqual (enumFromTo 0 2)
= let x = enumFromTo 0 2 in x == x
= let x = 0 : enumFromTo (0 + 1) 2 in x == x
= let x = 0 : x'; x' = enumFromTo (0 + 1) 2 in x == x
= let x = 0 : x'; x' = enumFromTo (0 + 1) 2
  in 0 == 0 && x' == x'
= let x' = enumFromTo (0 + 1) 2 in 0 == 0 && x' == x'
= let x' = enumFromTo (0 + 1) 2 in True && x' == x'
```

Equational reasoning

This is where sharing comes into play:

```
   selfEqual (enumFromTo 0 2)
 = let x = enumFromTo 0 2 in x == x
 = let x = 0 : enumFromTo (0 + 1) 2 in x == x
 = let x = 0 : x'; x' = enumFromTo (0 + 1) 2 in x == x
 = let x = 0 : x'; x' = enumFromTo (0 + 1) 2
   in 0 == 0 && x' == x'
 = let x' = enumFromTo (0 + 1) 2 in 0 == 0 && x' == x'
 = let x' = enumFromTo (0 + 1) 2 in True && x' == x'
 = let x' = enumFromTo (0 + 1) 2 in x' == x'
```

Equational reasoning

This is where sharing comes into play:

```
   selfEqual (enumFromTo 0 2)
 = let x = enumFromTo 0 2 in x == x
 = let x = 0 : enumFromTo (0 + 1) 2 in x == x
 = let x = 0 : x'; x' = enumFromTo (0 + 1) 2 in x == x
 = let x = 0 : x'; x' = enumFromTo (0 + 1) 2
   in 0 == 0 && x' == x'
 = let x' = enumFromTo (0 + 1) 2 in 0 == 0 && x' == x'
 = let x' = enumFromTo (0 + 1) 2 in True && x' == x'
 = let x' = enumFromTo (0 + 1) 2 in x' == x'
 = ...
 = True
```

Linear time, but constant space.

Top-level sharing

A somewhat special case is sharing introduced at the top-level.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)

expensive :: Int
expensive = fib 32
```

Top-level sharing

A somewhat special case is sharing introduced at the top-level.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)

expensive :: Int
expensive = fib 32
```

Sometimes referred to as **CAF (constant applicative form)**.

Top-level sharing

A somewhat special case is sharing introduced at the top-level.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)

expensive :: Int
expensive = fib 32
```

Sometimes referred to as **CAF (constant applicative form)**.

Can be immensely useful, but the lifetime of such an expression is potentially the entire run of the program.

Lightweight inspection

```
GHCi> x = [0 .. 2] :: [Int]
GHCi> :sprint x
x = _
GHCi> null x
False
GHCi> :sprint x
x = 0 : _
```

There is also `:print` which shows slightly more information.

Lightweight inspection

```
GHCi> x = [0 .. 2] :: [Int]
GHCi> :sprint x
x = _
GHCi> null x
False
GHCi> :sprint x
x = 0 : _
```

There is also `:print` which shows slightly more information.

Neither command works with cyclic structures. There are other tools such as **ghc-heap-view** or **ghc-debug** that are needed for inspecting those.

Example 4: map vs. reverse

Building a pipeline

```
example4a :: Int -> Bool
example4a n = null (map (<= 10) [0 .. n])
```

The new aspect compared to earlier examples is the addition of `map` in the middle of the pipeline – does it change anything?

Definition of `map`

```
map :: (a -> b) -> [a] -> [b]
map _ []       = []
map f (x : xs) = f x : map f xs
```

Equational reasoning

```
    null (map (<= 10) (enumFromTo 0 2))
```

Equational reasoning

```
null (map (<= 10) (enumFromTo 0 2))
```

Well-Typed

Equational reasoning

```
  null (map (<= 10) (enumFromTo 0 2))
= null (map (<= 10) (0 : enumFromTo (0 + 1) 2))
```

Equational reasoning

```
    null (map (<= 10) (enumFromTo 0 2))
  = null (map (<= 10) (0 : enumFromTo (0 + 1) 2))
```

Well-Typed

Equational reasoning

```
  null (map (<= 10) (enumFromTo 0 2))
= null (map (<= 10) (0 : enumFromTo (0 + 1) 2))
= null ((0 <= 10) : map (<= 10) enumFromTo (0 + 1) 2)
```

Well-Typed

Equational reasoning

```
  null (map (<= 10) (enumFromTo 0 2))
= null (map (<= 10) (0 : enumFromTo (0 + 1) 2))
= null ((0 <= 10) : map (<= 10) enumFromTo (0 + 1) 2)
```

Well-Typed

Equational reasoning

```
  null (map (<= 10) (enumFromTo 0 2))
= null (map (<= 10) (0 : enumFromTo (0 + 1) 2))
= null ((0 <= 10) : map (<= 10) enumFromTo (0 + 1) 2)
= False
```

Equational reasoning

```
  null (map (<= 10) (enumFromTo 0 2))
= null (map (<= 10) (0 : enumFromTo (0 + 1) 2))
= null ((0 <= 10) : map (<= 10) enumFromTo (0 + 1) 2)
= False
```

Still **constant space** (and time).

Adding a different function

```
example4b :: Int -> Bool
example4b n = null (reverse [0 .. n])
```

Definition of `reverse`

```haskell
reverse :: [a] -> [a]
reverse = reverseAcc []

reverseAcc :: [a] -> [a] -> [a]
reverseAcc acc []       = acc
reverseAcc acc (x : xs) = reverseAcc (x : acc) xs
```

Equational reasoning

```
  null (reverse (enumFromTo 0 2))
= null (reverseAcc [] (enumFromTo 0 2))
= null (reverseAcc [] (0 : enumFromTo (0 + 1) 2))
= null (reverseAcc (0 : []) (enumFromTo (0 + 1) 2))
```

Equational reasoning

```
  null (reverse (enumFromTo 0 2))
= null (reverseAcc [] (enumFromTo 0 2))
= null (reverseAcc [] (0 : enumFromTo (0 + 1) 2))
= null (reverseAcc (0 : []) (enumFromTo (0 + 1) 2))
= null (reverseAcc (0 : []) (1 : enumFromTo (1 + 1) 2))
= null (reverseAcc (1 : 0 : []) (enumFromTo (1 + 1) 2))
= null (reverseAcc (1 : 0 : []) (2 : enumFromTo (2 + 1) 2))
= null (reverseAcc (2 : 1 : 0 : []) (enumFromTo (2 + 1) 2))
= null (reverseAcc (2 : 1 : 0 : []) [])
= null (2 : 1 : 0 : [])
= False
```

Equational reasoning

```
   null (reverse (enumFromTo 0 2))
 = null (reverseAcc [] (enumFromTo 0 2))
 = null (reverseAcc [] (0 : enumFromTo (0 + 1) 2))
 = null (reverseAcc (0 : []) (enumFromTo (0 + 1) 2))
 = null (reverseAcc (0 : []) (1 : enumFromTo (1 + 1) 2))
 = null (reverseAcc (1 : 0 : []) (enumFromTo (1 + 1) 2))
 = null (reverseAcc (1 : 0 : []) (2 : enumFromTo (2 + 1) 2))
 = null (reverseAcc (2 : 1 : 0 : []) (enumFromTo (2 + 1) 2))
 = null (reverseAcc (2 : 1 : 0 : []) [])
 = null (2 : 1 : 0 : [])
 = False
```

This operates in **linear** space (and time).

Comparing `map` and `reverse`

What is the key difference between `map` and `reverse`?

Comparing `map` and `reverse`

What is the key difference between `map` and `reverse`?

The function `map` is **incremental**, while `reverse` is not.

Comparing `map` and `reverse`

What is the key difference between `map` and `reverse`?

The function `map` is **incremental**, while `reverse` is not.

More precisely:

- for `map`, we only need to evaluate the input list as far as we want to evaluate the output list.
- for `reverse`, even for just evaluating the result list to WHNF, we have to evaluate the entire spine of the input list.

Comparing `map` and `reverse`

What is the key difference between `map` and `reverse`?

The function `map` is **incremental**, while `reverse` is not.

More precisely:

- for `map`, we only need to evaluate the input list as far as we want to evaluate the output list.
- for `reverse`, even for just evaluating the result list to WHNF, we have to evaluate the entire spine of the input list.

Incrementality is not precisely defined, but I am calling functions incremental that can produce (parts of) their output without evaluating all of their input.

Incrementality

Which of the following functions are (or should be) incremental?

```
map f

reverse
```

Well-Typed

Incrementality

Which of the following functions are (or should be) incremental?

```
map f
reverse
filter p
```

Incrementality

Which of the following functions are (or should be) incremental?

```
map f
reverse
filter p
length
```

Well-Typed

Incrementality

Which of the following functions are (or should be) incremental?

```
map f
reverse
filter p
length
sum
```

Well-Typed

Incrementality

Which of the following functions are (or should be) incremental?

```
map f
reverse
filter p
length
sum
and
```

Well-Typed

Incrementality

Which of the following functions are (or should be) incremental?

```
map f
reverse
filter p
length
sum
and
take n
```

Well-Typed

Incrementality

Which of the following functions are (or should be) incremental?

```
map f
reverse
filter p
length
sum
and
take n
drop n
```

Example 5: length

Changing the definition of `null` once more

```haskell
nullViaLength :: [a] -> Bool
nullViaLength xs = length xs == 0

example5a :: Int -> Bool
example5a n = nullViaLength [0 .. n]
```

How does this compare to the other definitions of `null` ?

A simpler example

Let us just look at `length` itself:

```
example5b :: Int -> Int
example5b n = length [0 .. n]
```

What is the space behaviour?

Definition(s) of `length`

A (naive) definition of `length` is bad:

```
length :: [a] -> Int
length []       = 0
length (_ : xs) = 1 + length xs
```

Well-Typed

Equational reasoning

```
  length (enumFromTo 0 2)
= length (0 : enumFromTo (0 + 1) 2)
= 1 + length (enumFromTo (0 + 1) 2)
```

Equational reasoning

```
  length (enumFromTo 0 2)
= length (0 : enumFromTo (0 + 1) 2)
= 1 + length (enumFromTo (0 + 1) 2)
= 1 + length (1 : enumFromTo (1 + 1) 2)
= 1 + (1 + (length (enumFromTo (1 + 1) 2)))
```

Well-Typed

Equational reasoning

```
   length (enumFromTo 0 2)
 = length (0 : enumFromTo (0 + 1) 2)
 = 1 + length (enumFromTo (0 + 1) 2)
 = 1 + length (1 : enumFromTo (1 + 1) 2)
 = 1 + (1 + (length (enumFromTo (1 + 1) 2)))
 = ...
 = 1 + (1 + (1 + 0))
 = ...
 = 3
```

Equational reasoning

```
  length (enumFromTo 0 2)
= length (0 : enumFromTo (0 + 1) 2)
= 1 + length (enumFromTo (0 + 1) 2)
= 1 + length (1 : enumFromTo (1 + 1) 2)
= 1 + (1 + (length (enumFromTo (1 + 1) 2)))
= ...
= 1 + (1 + (1 + 0))
= ...
= 3
```

Runs in **linear** space.

Definition(s) of `length`

An accumulating definition of `length` is potentially not much better:

```
length :: [a] -> Int
length = lengthAcc 0

lengthAcc :: Int -> [a] -> Int
lengthAcc acc []       = acc
lengthAcc acc (_ : xs) = lengthAcc (1 + acc) xs
```

Equational reasoning

```
   length (enumFromTo 0 2)
 = lengthAcc 0 (enumFromTo 0 2)
 = lengthAcc 0 (0 : enumFromTo (0 + 1) 2)
 = lengthAcc (1 + 0) (enumFromTo (0 + 1) 2)
```

Well-Typed

Equational reasoning

```
  length (enumFromTo 0 2)
= lengthAcc 0 (enumFromTo 0 2)
= lengthAcc 0 (0 : enumFromTo (0 + 1) 2)
= lengthAcc (1 + 0) (enumFromTo (0 + 1) 2)
= lengthAcc (1 + 0) (1 : enumFromTo (1 + 1) 2)
= lengthAcc (1 + (1 + 0)) (enumFromTo (1 + 1) 2)
```

Equational reasoning

```
   length (enumFromTo 0 2)
 = lengthAcc 0 (enumFromTo 0 2)
 = lengthAcc 0 (0 : enumFromTo (0 + 1) 2)
 = lengthAcc (1 + 0) (enumFromTo (0 + 1) 2)
 = lengthAcc (1 + 0) (1 : enumFromTo (1 + 1) 2)
 = lengthAcc (1 + (1 + 0)) (enumFromTo (1 + 1) 2)
...
 = lengthAcc (1 + (1 + (1 + 0))) []
 = 1 + (1 + (1 + 0))
 = ...
 = 3
```

Also runs in **linear** space.

Definition(s) of `length`

We can fix the problem by artifically making `lengthAcc` more strict:

```
length :: [a] -> Int
length = lengthAcc 0

lengthAcc :: Int -> [a] -> Int
lengthAcc !acc []       = acc
lengthAcc !acc (_ : xs) = lengthAcc (1 + acc) xs
```

Definition(s) of `length`

We can fix the problem by artifically making `lengthAcc` more strict:

```
length :: [a] -> Int
length = lengthAcc 0

lengthAcc :: Int -> [a] -> Int
lengthAcc !acc []       = acc
lengthAcc !acc (_ : xs) = lengthAcc (1 + acc) xs
```

A **bang pattern** match will force the argument into WHNF, just as if it was a constructor match.

Equational reasoning

```
  length (enumFromTo 0 2)
= lengthAcc 0 (enumFromTo 0 2)
= lengthAcc 0 (0 : enumFromTo (0 + 1) 2)
= lengthAcc (1 + 0) (enumFromTo (0 + 1) 2)
```

Equational reasoning

```
  length (enumFromTo 0 2)
= lengthAcc 0 (enumFromTo 0 2)
= lengthAcc 0 (0 : enumFromTo (0 + 1) 2)
= lengthAcc (1 + 0) (enumFromTo (0 + 1) 2)
```

Equational reasoning

```
    length (enumFromTo 0 2)
  = lengthAcc 0 (enumFromTo 0 2)
  = lengthAcc 0 (0 : enumFromTo (0 + 1) 2)
  = lengthAcc (1 + 0) (enumFromTo (0 + 1) 2)
  = lengthAcc 1 (enumFromTo (0 + 1) 2)
```

Equational reasoning

```
  length (enumFromTo 0 2)
= lengthAcc 0 (enumFromTo 0 2)
= lengthAcc 0 (0 : enumFromTo (0 + 1) 2)
= lengthAcc (1 + 0) (enumFromTo (0 + 1) 2)
= lengthAcc 1 (enumFromTo (0 + 1) 2)
= lengthAcc 1 (1 : enumFromTo (1 + 1) 2)
```

Equational reasoning

```
  length (enumFromTo 0 2)
= lengthAcc 0 (enumFromTo 0 2)
= lengthAcc 0 (0 : enumFromTo (0 + 1) 2)
= lengthAcc (1 + 0) (enumFromTo (0 + 1) 2)
= lengthAcc 1 (enumFromTo (0 + 1) 2)
= lengthAcc 1 (1 : enumFromTo (1 + 1) 2)
= lengthAcc 2 (2 : enumFromTo (2 + 1) 2)
= lengthAcc 3 []
= 3
```

Now runs in **constant space** (but still linear time).

Aside: more on bang patterns

Note: **bang patterns only ever make sense on variables**.

(Why?)

Aside: `seq`

Historically, Haskell has had `seq` to control evaluation.

It is primitive, but you could define it in terms of bang patterns:

```
seq :: a -> b -> b
seq !_ y = y
```

Aside: `seq`

Historically, Haskell has had `seq` to control evaluation.

It is primitive, but you could define it in terms of bang patterns:

```
seq :: a -> b -> b
seq !_ y = y
```

```
lengthAcc :: Int -> [a] -> Int
lengthAcc acc []       = acc
lengthAcc acc (_ : xs) = seq acc (lengthAcc (1 + acc) xs)
```

Question about seq

Why not

```
force :: a -> a
force x = seq x x
```

```
lengthAcc :: Int -> [a] -> Int
lengthAcc acc []       = acc
lengthAcc acc (_ : xs) = lengthAcc (force (1 + acc)) xs
```

Question about `seq`

Why not

```
force :: a -> a
force x = seq x x
```

```
lengthAcc :: Int -> [a] -> Int
lengthAcc acc []       = acc
lengthAcc acc (_ : xs) = lengthAcc (force (1 + acc)) xs
```

`force` is just `id`. It does not create any demand that does not already exist.

Demand analysis

With optimisations on, GHC will detect that the original accumulating version of `length` will **always eventually use** the accumulator and make it strict even without bang pattern.

Yet another definition of `length`

```
length :: [a] -> Int
length = lengthAcc 0

lengthAcc :: Int -> [a] -> Int
lengthAcc _   []       = 0
lengthAcc acc [_]      = 1 + acc
lengthAcc acc (_ : xs) = lengthAcc (1 + acc) xs
```

This version does **not** always use `acc`, and therefore will not be optimised to use a strict accumulator.

Returning to our initial example

```
nullViaLength :: [a] -> Bool
nullViaLength xs = length xs == 0

example5a :: Int -> Bool
example5a n = nullViaLength [0 .. n]
```

Well-Typed

Returning to our initial example

```
nullViaLength :: [a] -> Bool
nullViaLength xs = length xs == 0

example5a :: Int -> Bool
example5a n = nullViaLength [0 .. n]
```

Constant space, but linear time, and therefore unsuitable as a definition of `null` .

Another variant

```
if nullViaLength xs
  then ...
  else ... sum xs ...
```

Well-Typed

Another variant

```
if nullViaLength xs
  then ...
  else ... sum xs ...
```

Sharing can turn something that just looks unnecessarily inefficient into a space leak.

Example 6: unfair partitioning

Partitioning a list

```
example6 :: Int -> (Int, Int)
example6 n =
  case partition (>= 0) [0 .. n] of
    (xs, ys) -> (sum xs, sum ys)
```

(Think of `(>= 0)` as some kind of sanity check.)

Defining `partition`

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
partition _ []       = ([], [])
partition p (x : xs) =
  case partition p xs of
    (ys, zs)
      | p x       -> (x : ys, zs)
      | otherwise -> (ys, x : zs)
```

Is this a good definition?

Equational reasoning

```
   partition (>= 0) (enumFromTo (0 .. 2))
 = partition (>= 0) (0 : enumFromTo (0 + 1) 2)
 = case partition (>= 0) (enumFromTo (0 + 1) 2) of
     (ys, zs)
        | (>= 0) 0   -> (0 : ys, zs)
        | otherwise -> (ys, 0 : zs)
```

Equational reasoning

```
  partition (>= 0) (enumFromTo (0 .. 2))
= partition (>= 0) (0 : enumFromTo (0 + 1) 2)
= case partition (>= 0) (enumFromTo (0 + 1) 2) of
    (ys, zs)
       | (>= 0) 0  -> (0 : ys, zs)
       | otherwise -> (ys, 0 : zs)
= ...
= case (case partition (>= 0) (enumFromTo (1 + 1) 2) of
          (ys', zs')
             | (>= 0) 1  -> (1 : ys, zs)
             | otherwise -> (ys, 1 : zs)
       ) of
    (ys, zs)
       | (>= 0) 0  -> (0 : ys, zs)
       | otherwise -> (ys, 0 : zs)
```

Oh no …

Irrefutable pattern matches

We **know** the result of partition will be a pair, so why wait?

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
partition _ []          = ([], [])
partition p (x : xs) =
  case partition p xs of
    ~(ys, zs)
      | p x       -> (x : ys, zs)
      | otherwise -> (ys, x : zs)
```

An **irrefutable** match will always succeed. You can think of it as being rewritten to using selectors.

An equivalent but uglier definition of `partition`

```haskell
partition :: (a -> Bool) -> [a] -> ([a], [a])
partition _ [] = ([], [])
partition p (x : xs) =
  let r = partition p xs
  in  if p x
        then (x : fst r, snd r)
        else (fst r, x : snd r)
```

Well-Typed

Aside: irrefutable patterns

Why are irrefutable patterns so rare?

Aside: irrefutable patterns

Why are irrefutable patterns so rare?

Because `let` pattern matches are implicitly irrefutable.

Aside: irrefutable patterns

Why are irrefutable patterns so rare?

Because `let` pattern matches are implicitly irrefutable.

Can you think of other functions that morally require an irrefutable pattern match?

Equational reasoning

```
  partition (>= 0) (enumFromTo (0 .. 2))
= partition (>= 0) (0 : enumFromTo (0 + 1) 2)
= let r = partition (>= 0) (enumFromTo (0 + 1) 2)
  in  if (>= 0) 0
        then (0 : fst r, snd r)
        else (fst r, 0 : snd r)
```

Equational reasoning

```
  partition (>= 0) (enumFromTo (0 .. 2))
= partition (>= 0) (0 : enumFromTo (0 + 1) 2)
= let r = partition (>= 0) (enumFromTo (0 + 1) 2)
  in  if (>= 0) 0
        then (0 : fst r, snd r)
        else (fst r, 0 : snd r)
= let r = partition (>= 0) (enumFromTo (0 + 1) 2)
  in  (0 : fst r, snd r)
```

This is better. We already have quite a bit of information at this point – in particular, the result is now in WHNF!

Equational reasoning

Let's assume we place more demand on the first component of the result pair, i.e., on `fst r` :

```
    let r = partition (>= 0) (enumFromTo (0 + 1) 2)
    in  (0 : fst r, snd r)
```

Equational reasoning

Let's assume we place more demand on the first component of the result pair, i.e., on `fst r` :

```
   let r = partition (>= 0) (enumFromTo (0 + 1) 2)
   in  (0 : fst r, snd r)
 = let r = let r' = partition (>= 0) (enumFromTo (1 + 1) 2)
           in  (1 : fst r', snd r')
   in  (0 : fst r, snd r)
```

Equational reasoning

Let's assume we place more demand on the first component of the result pair, i.e., on `fst r` :

```
   let r = partition (>= 0) (enumFromTo (0 + 1) 2)
   in  (0 : fst r, snd r)
 = let r = let r' = partition (>= 0) (enumFromTo (1 + 1) 2)
           in  (1 : fst r', snd r')
   in  (0 : fst r, snd r)
 = let r' = partition (>= 0) (enumFromTo (1 + 1) 2)
       r  = (1 : fst r', snd r')
   in  (0 : fst r, snd r)
```

Equational reasoning

Let's assume we place more demand on the first component of the result pair, i.e., on `fst r` :

```
   let r = partition (>= 0) (enumFromTo (0 + 1) 2)
   in  (0 : fst r, snd r)
 = let r = let r' = partition (>= 0) (enumFromTo (1 + 1) 2)
           in  (1 : fst r', snd r')
   in  (0 : fst r, snd r)
 = let r' = partition (>= 0) (enumFromTo (1 + 1) 2)
       r  = (1 : fst r', snd r')
   in  (0 : fst r, snd r)
 = let r' = partition (>= 0) (enumFromTo (1 + 1) 2)
       r  = (1 : fst r', snd r')
   in  (0 : 1 : fst r', snd r)
```

Isn't there still a problem here?

Selector thunk optimisation

```
   let r' = partition (>= 0) (enumFromTo (1 + 1) 2)
       r  = (1 : fst r', snd r')
   in  (0 : 1 : fst r', snd r)
 = let r' = partition (>= 0) (enumFromTo (1 + 1) 2)
   in  (0 : 1 : fst r', snd r')
```

The **garbage collector** will reduce **selector thunks** if possible, even if there's no explicit demand on them.

Revisiting the example

```haskell
example6 :: Int -> (Int, Int)
example6 n =
  case partition (>= 0) [0 .. n] of
    (xs, ys) -> (sum xs, sum ys)
```

Well-Typed

Equational reasoning

```
case partition (>= 0) (enumFromTo 0 2) of
  (xs, ys) -> (sum xs, sum ys)
```

Equational reasoning

```
   case partition (>= 0) (enumFromTo 0 2) of
     (xs, ys) -> (sum xs, sum ys)
 = case (let r = partition (>= 0) (enumFromTo (0 + 1) 2)
         in  (0 : fst r, snd r)) of
     (xs, ys) -> (sum xs, sum ys)
```

Equational reasoning

```
  case partition (>= 0) (enumFromTo 0 2) of
    (xs, ys) -> (sum xs, sum ys)
= case (let r = partition (>= 0) (enumFromTo (0 + 1) 2)
        in  (0 : fst r, snd r)) of
    (xs, ys) -> (sum xs, sum ys)
= let r  = partition (>= 0) (enumFromTo (0 + 1) 2)
  in  (sum (0 : fst r), sum (snd r))
```

This is in WHNF. Will it be ok if we proceed placing demand on it, e.g. by printing the result?

Example 7: fair partitioning

A variant of our previous example

```
example7a :: Int -> (Int, Int)
example7a n =
  case partition even [0 .. n] of
    (xs, ys) -> (sum xs, sum ys)
```

The only difference is that we are using `even` instead of `(>= 0)`.

Equational reasoning

```
    case partition even (enumFromTo 0 2) of
      (xs, ys) -> (sum xs, sum ys)
```

Equational reasoning

```
    case partition even (enumFromTo 0 2) of
      (xs, ys) -> (sum xs, sum ys)
 =  case (let r = partition even (enumFromTo (0 + 1) 2)
          in  (0 : fst r, snd r)) of
      (xs, ys) -> (sum xs, sum ys)
```

Equational reasoning

```
   case partition even (enumFromTo 0 2) of
     (xs, ys) -> (sum xs, sum ys)
 = case (let r = partition even (enumFromTo (0 + 1) 2)
         in  (0 : fst r, snd r)) of
     (xs, ys) -> (sum xs, sum ys)
 = let r  = partition even (enumFromTo (0 + 1) 2)
   in  (sum (0 : fst r), sum (snd r))
```

Equational reasoning

```
   case partition even (enumFromTo 0 2) of
     (xs, ys) -> (sum xs, sum ys)
 = case (let r = partition even (enumFromTo (0 + 1) 2)
         in  (0 : fst r, snd r)) of
     (xs, ys) -> (sum xs, sum ys)
 = let r  = partition even (enumFromTo (0 + 1) 2)
   in  (sum (0 : fst r), sum (snd r))
 = let r  = partition even (enumFromTo (1 + 1) 2)
   in  (sumAcc 0 (fst r), sum (1 : snd r))
```

While we are evaluating the first component of the pair, the second component grows larger …

A better way?

The problematic pattern here is that we are generating

```
([Int], [Int])
```

but the generation of the two lists is not independent, and the distribution is not statically known.

A better way?

The problematic pattern here is that we are generating

```
([Int], [Int])
```

but the generation of the two lists is not independent, and the distribution is not statically known.

```
partitionEvenSums :: [Int] -> (Int, Int)
partitionEvenSums = partitionEvenSumsAcc (0, 0)

partitionEvenSumsAcc :: (Int, Int) -> [Int] -> (Int, Int)
partitionEvenSumsAcc (!x, !y) []       = (x, y)
partitionEvenSumsAcc (!x, !y) (z : zs) =
  if even z then partitionEvenSumsAcc (x + z, y) zs
            else partitionEvenSumsAcc (x, y + z) zs
```

Revisiting the example

```
example7b :: Int -> (Int, Int)
example7b n = partitionEvenSums [0 .. n]
```

This works in **constant space** (but is less modular).

Revisiting the example

```
example7b :: Int -> (Int, Int)
example7b n = partitionEvenSums [0 .. n]
```

This works in **constant space** (but is less modular).

Libraries such as **foldl** or **streamly** can help restore modularity here.

Writer monad

```haskell
data Writer w a = Writer w a
```

A similar problem arises here as we have seen for partitioning. For `Writer`, it is typically even worse because monadic computations will often run for a very long time.

Example 8: effectful traversals

Traversing a list

```
example8a n = length <$> traverse pure [0 .. n]
```

Traversing a list

```haskell
example8a n = length <$> traverse pure [0 .. n]
```

Definition of `traverse` on lists:

```haskell
traverse :: Applicative f => (a -> f b) -> [a] -> f [b]
traverse _ []       = pure []
traverse f (x : xs) = pure (:) <*> f x <*> traverse f xs
```

What applicative functor?

Does the choice of applicative functor matter?

What applicative functor?

Does the choice of applicative functor matter?

What about each of

- `Identity`
- `Maybe`
- `IO`

Identity

```
example8a :: Int -> Identity Int

newtype Identity a = Identity {runIdentity :: a}

instance Functor Identity where
  fmap f x = pure f <*> x

instance Applicative Identity where
  pure = Identity
  f <*> x = Identity ((runIdentity f) (runIdentity x))
```

Equational reasoning

```
   traverse pure (enumFromTo 0 2)
=  traverse pure (0 : enumFromTo (0 + 1) 2)
=  pure (:) <*> pure 0
      <*> traverse pure (enumFromTo (0 + 1) 2)
```

Equational reasoning

```
    traverse pure (enumFromTo 0 2)
  = traverse pure (0 : enumFromTo (0 + 1) 2)
  = pure (:) <*> pure 0
       <*> traverse pure (enumFromTo (0 + 1) 2)
  = Identity (runIdentity (pure (:)) <*> runIdentity (pure 0))
       <*> traverse pure (enumFromTo (0 + 1) 2)
```

Equational reasoning

```
     traverse pure (enumFromTo 0 2)
  =  traverse pure (0 : enumFromTo (0 + 1) 2)
  =  pure (:) <*> pure 0
       <*> traverse pure (enumFromTo (0 + 1) 2)
  =  Identity (runIdentity (pure (:)) <*> runIdentity (pure 0))
       <*> traverse pure (enumFromTo (0 + 1) 2)
  =  Identity ((:) 0)
       <*> traverse pure (enumFromTo (0 + 1) 2)
```

Equational reasoning

```
   traverse pure (enumFromTo 0 2)
 = traverse pure (0 : enumFromTo (0 + 1) 2)
 = pure (:) <*> pure 0
      <*> traverse pure (enumFromTo (0 + 1) 2)
 = Identity (runIdentity (pure (:)) <*> runIdentity (pure 0))
      <*> traverse pure (enumFromTo (0 + 1) 2)
 = Identity ((:) 0)
      <*> traverse pure (enumFromTo (0 + 1) 2)
 = Identity
     (0 : runIdentity (traverse pure (enumFromTo (0 + 1) 2)))
```

This looks fine (and it is).

Runs in **constant space**.

Maybe

```haskell
example8b :: Int -> Maybe Int
```

```haskell
data Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap f x = pure f <*> x

instance Applicative Maybe where
  pure = Just
  Nothing <*> _       = Nothing
  Just _  <*> Nothing = Nothing
  Just f  <*> Just x  = Just (f x)
```

Equational reasoning

```
traverse pure (enumFromTo 0 2)
 = traverse pure (0 : enumFromTo (0 + 1) 2)
 = pure (:) <*> pure 0
     <*> traverse pure (enumFromTo (0 + 1) 2)
 = Just (:) <*> Just 0
     <*> traverse pure (enumFromTo (0 + 1) 2)
 = Just ((:) 0) <*> traverse pure (enumFromTo (0 + 1) 2)
```

This is looking bad.

Runs in **linear space**.

A possible fix

```haskell
traverseLength :: [a] -> Maybe Int
traverseLength = traverseLengthAcc 0

traverseLengthAcc :: Int -> [a] -> Maybe Int
traverseLengthAcc !acc [] = Just acc
traverseLengthAcc !acc (x : xs) =
  pure x *> traverseLengthAcc (1 + acc) xs
```

Conclusions